Instructor's Manual

MOTOROLA PowerPC

Excimer Laboratory Manual

Jose I. Quiñones, Noel Serrano, Walter Guiot, Luis Narváez, Eisen Montalvo
Department of Electrical and Computer Engineering
University of Puerto Rico-Mayagüez

Chuck Corley
PowerPC Applications Engineering
Motorola

Editor: José L. Cruz Rivera
Department of Electrical and Computer Engineering
University of Puerto Rico-Mayagüez

VOLUME I

DISCLAIMERS

This document contains information on a new product under development by Motorola and IBM. Motorola and IBM reserve the right to change or discontinue this product without notice. Information in this document is provided solely to enable system and software implementers to use PowerPC microprocessors. There are no express or implied copyright or patent licenses granted hereunder by Motorola or IBM to design, modify the design of, or fabricate circuits based on the information in this document. The PowerPC 60x microprocessors embody the intellectual property of Motorola and of IBM. However, neither Motorola nor IBM assumes any responsibility or liability as to any aspects of the performance, operation, or other attributes of the microprocessor as marketed by the other party or by any third party. Neither Motorola nor IBM is to be considered an agent or representative of the other, and neither has assumed, created, or granted hereby any right or authority to the other, or to any third party, to assume or create any express or implied obligations on its behalf. Information such as data sheets, as well as sales terms and conditions such as prices, schedules, and support, for the product may vary as between parties selling the product. Accordingly, customers wishing to learn more information about the products as marketed by a given party should contact that party. Both Motorola and IBM reserve the right to modify this manual and/or any of the products as described herein without further notice.

NOTHING IN THIS MANUAL, NOR IN ANY OF THE ERRATA SHEETS, DATA SHEETS, AND OTHER SUPPORTING DOCUMENTATION, SHALL BE INTERPRETED AS THE CONVEYANCE BY MOTOROLA OR IBM OF AN EXPRESS WARRANTY OF ANY KIND OR IMPLIED WARRANTY, REPRESENTATION, OR GUARANTEE REGARDING THE MERCHANTABILITY OR FITNESS OF THE PRODUCTS FOR ANY PARTICULAR PURPOSE.

Neither Motorola nor IBM assumes any liability or obligation for damages of any kind arising out of the application or use of these materials. Any warranty or other obligations as to the products described herein shall be undertaken solely by the marketing party to the customer, under a separate sale agreement between the marketing party and the customer. In the absence of such an agreement, no liability is assumed by Motorola, IBM, or the marketing party for any damages, actual or otherwise. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals," must be validated for each customer application by customer's technical experts. Neither Motorola nor IBM convey any license under their respective intellectual property rights nor the rights of others. Neither Motorola nor IBM makes any claim, warranty, or representation, express or implied, that the products described in this manual are designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the product could create a situation where personal injury or death may occur. Should customer purchase or use the products for any such unintended or unauthorized application, customer shall indemnify and hold Motorola and IBM and their respective officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola or IBM was negligent regarding the design or manufacture of the part.

Motorola and are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer. IBM and IBM logo are registered trademarks, and IBM Microelectronics is a trademark of International Business Machines Corp. The PowerPC name, PowerPC logotype, and PowerPC 601 are trademarks of International Business Machines Corp. used by Motorola under license from International Business Machines Corp. International Business Machines Corp. is an Equal Opportunity/Affirmative Action Employer.

INTRODUCTION

This laboratory manual contains 13 lab experiments for the PowerPC Excimer Board presented in an increasing order of complexity. The experiments range from memory mapping problems and system benchmarking to integer to floating point number representation conversion. It is assumed that the student has a basic understanding of C and assembly languages. There is a natural progression in the lab experiments leading up to the Dhrystone and Linpack benchmarking of the PowerPC603e that forms the basis of the Excimer board. Specifically, the experiments guide the student through the following topics: code compilation, code download, DINK functions (resident monitor program), keyboard input, assembly language programming, and linking assembly language to C code. There are also experiments on memory mapping and Flash ROM programming.

Each lab experiment is structured as follows: Problem Statement, Objectives, Background Information, Procedure, Questions, and References. The *Problem Statement* provides a brief indication as to the tasks that will be performed. The *Objectives* section presents the specific educational objectives that will be met upon successful completion of the lab experiment. The *Background* information section presents a brief description of the theory behind the devices, instructions, functional units, and/or methods to be followed in the conduction of the experiment. The *Procedure* section presents a step-by-step guide to the experiment. The *Questions* section seeks to guide the student through a meaningful analysis of what he/she has performed as part of the experiment. Finally, the *References* section presents additional references with material that is useful for the experiment at hand. In addition to these sections, the Instructor's Manual contains a *Results* and a *Troubleshooting* section.

This laboratory manual contains experiments designed to familiarize students with the PowerPC architecture via the Excimer Laboratory Board. The lab manual is not meant to serve as a stand-alone textbook on the PowerPC instruction set architecture (ISA), but rather is designed as a companion to any PowerPC book or technical reference. Each experiment is designed so that students will end up with a significant number of useful subroutines that can be used in other more complex programming problems. Additional references to the PowerPC architecture and the Excimer board may be found at http://www.motorola.com/SPS/PowerPC/teksupport.

CONTENTS

Experiment

1

Metaware Tutorial

Problem Statement:

- In this experiment the student will develop and compile a C program that will calculate the first 12 Fibonacci Numbers using the Metaware PowerPC compiler. (Contributed by Noel Serrano).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- write, debug and compile a C program using the Metaware and Code Warrior compilers
- write a recursive function that will generate the first 12 Fibonacci Numbers

Background Information:

This experiment is designed to take you through the major steps required to implement a simple algorithm for the generation of the first 12 Fibonacci numbers using the Metaware compilers for the Excimer board. The Metaware compiler facilitates code writing, debugging, and optimization. More information on the compiler may be obtained from www.metaware.com.

The Fibonacci sequence represents a series that has as its first two elements 0 and 1. The remaining elements can be obtained by simply adding the last two numbers to get the next. For example, the first 12 Fibonacci Numbers (the first element in the sequence, 0, is not included) are:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

The Fibonacci numbers arose from the solution to the following problem posed in the year 1225: Suppose we have one pair of rabbits that can produce another pair of productive offspring when they reach the age of 1 month and that each successive pair of offspring can do the same. Furthermore, assume the rabbits never die. How many rabbits will there be after n months? The solution is as follows: If after n months there are k_n pairs of rabbits, the number of pairs in month $n+1$ will be k_n plus the number of new pairs born. However, since new pairs are only born to pairs at least 1 month old, there will be k_{n-1} new pairs, that is $k_{n+1} = k_n + k_{n-1}$, which is simply the rule for generating the Fibonacci Numbers. More information on the fascinating world of Fibonacci Numbers and their applications can be found in http://pass.maths.org.uk/issue3/fibonacci/index.html.

Procedure:

1. Write a C language program that will calculate the first 12 Fibonacci numbers.

   Hint: Use a recursive function.

2. To be able to print the numbers in the DINK32 interface (this will be discussed in more detail in future experiments) you will need to add the following code to your program to redefine the default printf function with the one provided by DINK. Also the printf function **should** contain only one variable.

```
#define printf dink_printf
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6270;
```

3. Type your program to a text file using *notepad* or *edit* and save it in the directory you have chosen to contain your code.

4. Compile the C code with the `hcppc` command included with the ***Metaware*** C compiler using the following command on the DOS command prompt :

```
hcppc -Hppc603 -c file.c
```

Note: "file.c" stands for the C code file. You may name your C code file as you wish, but remember to use the chosen name in the `hcppc` command. The result from this command will be "file.o", which is the object file. For more information about the options of the compiler type `hcppc -h`.

5. Link the object files using `ldppcl` command to invoke the linker program included with ***Metaware*** C compiler using the following command on the DOS Command Prompt:

```
ldppc -B start_addr=70000 -xm file.o
```

Note: "file.o" is the object file generated in the last step. The object file will be named exactly as you named the C code file.) The result from this step will be the file "a.hex", the file that will be later downloaded to the Excimer board. The `-B start_addr=70000` is an option that specifies where does your code is going to be paced in the memory of the Excimer Board. For more information about the linker type `ldppc -h`.

References:

- Metaware High C/C++ Compiler – http://www.metaware.com

Suggested Code:

```
int fibonacci(int x);

#define  printf dink_printf
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6270;

main()
{
    int fib_no = 0, index = 0;
    while (index < 12)
    {
        fib_no = fibonacci(index);
        printf("Fibonacci number for index %d is", index);
```

```
        printf(" %d\n", fib_no);
        index++;
    }
    return 0;
}

int fibonacci(int number)
{
    switch (number)
    {
       case 0 :
            return 1;
       case 1 :
            return 1;
       default :
            return (fibonacci(number-1) + fibonacci(number-2));
    }
}
```

Troubleshooting:

If the student is not able to:

- Print to the DINK 32 interface: verify that the address of the pointer address matches that of the DINK version in use. This can be done through an `st` command in DINK and verifying that the address for `printf` matches the one provided in this manual.

Experiment

2

DINK Tutorial

Problem Statement:

- This experiment is designed to introduce the student to the DINK interface. A tutorial on how to download code to the Excimer board and some useful DINK debugging utilities are also presented. (Contributed by Noel Serrano).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- download their programs to the Excimer board using the DINK interface.
- debug the programs using the DINK built-in debugging tools.

Background Information:

The Excimer board contains a debugging interface called DINK. This interface enables you to connect to the evaluation board through a serial cable using a terminal program. This enables the developer to have continuous communication with the evaluation board, allowing insight into the board's state at all time. The terminal screen of your program should look like this.

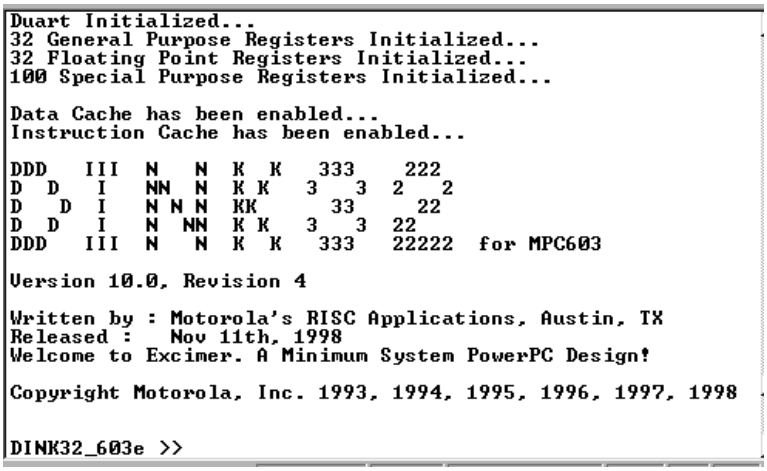


Figure 1. DINK32 on terminal client

More information on DINK can be found at www.mot.com/SPS/PowerPC/teksupport/teklibrary/index.html.

Procedure:

1. First make sure you have your evaluation board connected using the serial cable provided to serial port 1 (COM1). Open your terminal client and configure it to connect through COM1 using the following parameters.

Parameter	Value
Protocol	Serial
Port	COM1
Baud Rate	9600
Data Bits	8
Parity	N
Stop Bits	1
RTS/CTS	Enabled

   Turn on the evaluation board and press Connect on your terminal client. You should be able to see the initialization window with the `DINK32_603e >>` prompt as presented in the figure shown below (see step 4).

2. Compile the program you created on experiment number one using the Metaware compiler.

3. Now you are ready to download your program to the Excimer board for execution. To do so first go to the terminal client running the DINK32 interface and type `dl -k`. This command will expect to receive data from the keyboard serial port (COM1). Now proceed to send the file from the terminal client. This can be done by selecting a command like Send Text File or Send ASCII (this can vary from one terminal client to the other). Now browse for the `a.hex` created in the directory where you compiled your program.

4. Run your program by typing `go 70000` in the DINK32 program. If your code is correct and if you have been successful in downloading the code you should get an output like the following.

```
DINK32_603e >>dl -k
set to Keyboard Port
Download Complete.
DINK32_603e >>go 70000
Fibonacci number for index 0 is 1
Fibonacci number for index 1 is 1
Fibonacci number for index 2 is 2
Fibonacci number for index 3 is 3
Fibonacci number for index 4 is 5
Fibonacci number for index 5 is 8
Fibonacci number for index 6 is 13
Fibonacci number for index 7 is 21
Fibonacci number for index 8 is 34
Fibonacci number for index 9 is 55
Fibonacci number for index 10 is 89
Fibonacci number for index 11 is 144
DINK32_603e >>
```

Hint: The table below presents some useful commands in case you need to debug your program, view memory or register contents, and/or ser breakpoints for program tracing. For more information type `help <command>` in the DINK prompt.

Command	Format	Description
Memory Display	md <addr>	Displays the memory area specified by the hex address
Registry Display	regdisp rx	Displays the register specified by rx
Disassemble	ds <addr>	Disassemble the code starting at the specified address location
Trace	tr <addr>	Begin tracing a program at the specified address. To continue tracing type tr +.
Breakpoint	br <addr>	Sets a breakpoint at the specified address
Assemble	as <addr>	Provides you with the option of changing part of the assembly from the DINK Interface accessing it through the address of the code line.

References:

[1] Motorola, Designing a Minimal PowerPC System, PowerPC Application Note: AN1769/D, 1998.

Conclusions:

Students should be able to note that:

- DINK works similarly to other evaluation board environments
- DINK provides functionality that enables the user to modify the memory, registers, and assembly code
- DINK provides breakpoint and trace capabilities for debugging purposes

Troubleshooting:

If the student is not able to communicate with the DINK:

- verify the connections to COM1 port and board
- check for correct settings on terminal client

Experiment

3

Useful DINK Functions

Problem Statement:

- In this experiment the student is introduced to a set of useful functions that are contained within the DINK 32 Interface. (Contributed by Noel Serrano).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- work with more advanced DINK functions and use them on future laboratories.

Background Information:

The DINK 32 interface provides a set of functions that facilitate the development of programs for the Excimer board. Among the functions included in DINK are some that allow the programmer to capture data from the keyboard and to print to the screen. Other functions control parts of the Excimer board configuration like enabling the timer, cache, etc.

This laboratory will give you and overview of a basic set of these functions and will teach you how to access them in your C programs. There is a command included in DINK that display a list of all these functions with their branch labels and corresponding addresses. The command is `st` and the corresponding output will look like this.

```
DINK32_603e >>st
```

```
Current list of DINK branch labels:
         KEYBOARD:    0x0
         get_char:    0x1e4c4
       write_char:    0x5eb4
        TBaseInit:    0x39e0
   TBaseReadLower:    0x3a04
   TBaseReadUpper:    0x3a20
     CacheInhibit:    0x3a3c
    InvEnL1Dcache:    0x3a5c
      DisL1Dcache:    0x3aa4
    InvEnL1Icache:    0x3ac8
      DisL1Icache:    0x3b00
        BurstMode:    0x3bfc
         RamInCBk:    0x3c3c
       RamInWThru:    0x3c7c
        dink_loop:    0x55e8
      dink_printf:    0x6270

Current list of USER branch labels:
DINK32_603e >>
```

All these functions can be accessed through your C code by casting a function that will point to the address in DINK. The code that defines the function would look like the following example for the printf function.

```
#define  printf dink_printf
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6270;
```

In the following section we present examples of three DINK functions.

a) get_char – This function enables the programmer to capture characters from the keyboard through the DINK interface. The get_cahr function can be accessed by using the following code:

```
#define  getchar dink_get_char
unsigned long (*dink_get_char)() = (unsigned long (*)()) 0x1e4c4;
```

This will enable you to capture characters from the keyboard. The syntax for reading character from the keyboard would be:

```
char LED;
LED = getchar();
```

b) write_char – This function enables the programmer to display characters on the terminal screen that is running DINK. The write_char function can be accesed by using the following code.

```
#define  writechar dink_write_char
unsigned long (*dink_write_char)() = (unsigned long (*)())0x5eb4;
```

This will enable you to output characters to the screen. The sysntax for displaying single characters from the keyboard would be:

```
char LED = 'N';
LED = writechar(LED);
```

c) dink_printf –This functions provide the programmer the option of displaying a string of characters on the DINK interface and also provide the user the ability of including a runtime variable, either char or integer, on this string. It is done by using the dink_printf function using the same syntax as in C.

```
#define  printf dink_printf
unsigned long (*dink_printfr)() = (unsigned long (*)())0x6270;
```

This will enable you to print any message on the DINK and also include any of the variables included in your code. The DINK `printf` function can only include ***one*** variable per statement not like in C which it can contain any number of variables.

```
printf("Fibonacci number for index %d is", index);
```

There are other important functions that can be used to control many aspects of the Excimer board. These are briefly described in the table below and explained in details in the included DINK manuals.

Functions	Address	Description
TBaseInit	0x39e0	Initializes the time base register
TBaseReadLower	0x3a04	Reads the lower half of the time base register

TBaseReadUpper	0x3a20	Reads the upper half of the time base register.
CacheInhibit	0x3a3c	Turns off the caches.
InvEnL1Dcache	0x3a5c	Invalidate and Enable the L1 data cache.
DisL1Dcache	0x3aa4	Disable L1 data cache.
InvEnL1Icache	0x3ac8	Invalidate and Enable the L1 instruction cache.
DisL1Icache	0x3b00	Disable L1 instruction cache.
BurstMode	0x3bfc	Sets up burst mode.

References:

[1] Motorola, Designing a Minimal PowerPC System, PowerPC Application Note: AN1769/D, 1998.

Suggested Code:

/* This section of code can be used to define any of the DINK functions in a C language program. The user will only need to modyfy the address and function name. */

```
#define  function_name dink_function_name
unsigned long (*dink_function_name)() = (unsigned long (*)()) hex_addr;
```

Troubleshooting:

If the student is not able to access the DINK functions:

- verify the casting is correct.
- verify that he/she is using the correct hex address.

Experiment

4

Excimer Memory Map

Problem Statement:

- This experiment requires the compilation, downloading, and execution of a C language program which blinks the Excimer Board's STATUS and ERROR Light Emitting Diodes (LEDs). (Contributed by Noel Serrano, José I. Quiñones, Luis Naváez, Walter Guiot, and Gunther Costas).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- write and compile a C program
- download and execute ***PowerPC*** Assembly object code
- locate the LEDs within the ***Excimer's*** memory map
- apply the methodology needed to turn on and off the LEDs

Background Information:

The PowerPC family of microprocessors is based on a memory mapped input/output scheme. Under this scheme, an input port can be thought of as read-only memory location, while an output port can be treated like a write-only memory location. The microprocessor's address bus is used to select the peripheral device (port location), the data bus is used to transmit or receive data to/from the device, and the Transfer Type signals are used to convey the directionality of the information transfer.

The memory map for the Excimer Board is shown in Figure 2. The memory map indicates that out of a total of 2^{32} = 4GB addressable locations, the Excimer Board allocates 2^{30} = 1GB each to Static RAM, Fast I/O devices, Slow I/O devices and Flash ROM [1]. Of course, the Excimer board only uses a fraction of the memory locations allocated for each type of memory and devices. The Excimer Board is configured with 512 KBs of SRAM, 4MBs of Flash ROM, and some LED indicators. For example, there's a STATUS LED located at 0x40200000, while an ERROR LED is specified at 0x40600000.

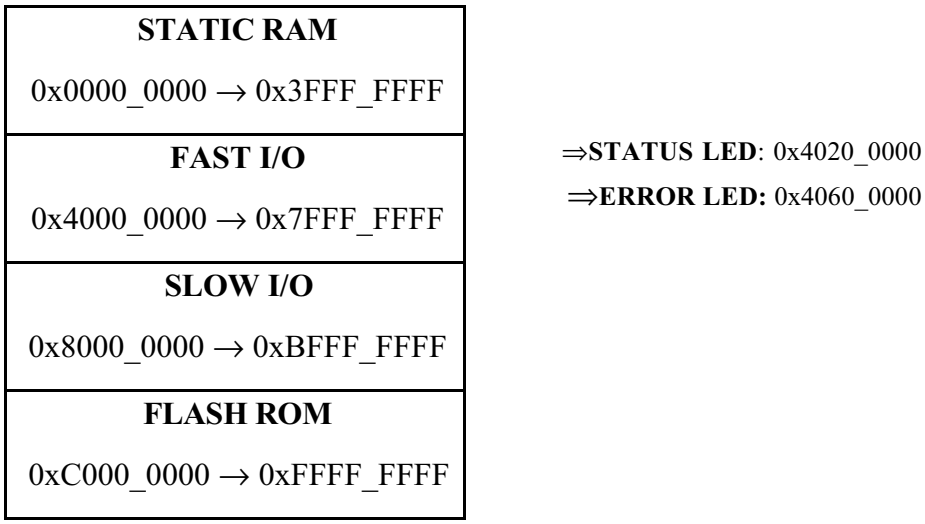


Figure 1: Excimer's Memory Map.

In this experiment you are required to write a C program that will blink (repeatedly turn on and off) the STATUS and ERROR LEDs alternatively. The LEDs are turned on/off by clearing/setting BIT 3 (fourth least significant bit) of these locations. The reason for this negative logic is that the LEDs are connected in a common anode configuration, as shown in Figure 2 for the case of a seven segment LED display.

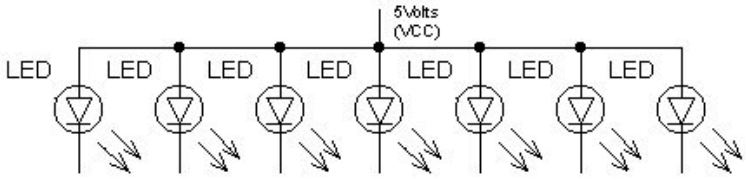


Figure 2: Common Anode LED configuration. LEDs will turn ON when the cathode is at ground level (*Excimer* Output asserted low)

To successfully blink an LED, you must carefully select the delay timing. Remember that the microprocessor may turn the LED on and off so quickly that you will not see the blinking effect. Since your program will be written in C, a simple "for" loop instruction may do the job.

```
For (counter=0;counter <= parameter; counter++);
```

Note: *counter* must be declared as *unsigned long* in the program. The value *parameter* will define the delay time.

There are other ways to accomplish a time delay, for example using the ***PowerPC's*** internal timer register. These techniques will be demonstrated in the successive experiments.

Procedure:

1. Write a simple C code that alternatively blinks the *Status* and *Error* LEDs ten times.

2. Compile the C code with the `hcppc` command included with the ***Metaware*** C compiler using the following command on the DOS command prompt :

   ```
   hcppc –Hppc603 –c file.c
   ```

Note: "File.c" stands for the C code file. You may name your C code file as you wish, but remember to use the chosen name in the `hcppc` command. The result from this command will be "file.o", which is the object file.

3. Link the object files using `ldppcl` command to invoke the linker program included with ***Metaware*** C compiler using the following command on the DOS Command Prompt:

   ```
   ldppc -B start_addr=70000 -xm file.o
   ```

   Note: "File.o" is the object file generated in the last step. The object file will be named exactly as you named the C code file.) The result from this step will be the file "a.hex".

4. Run the ***DINK32*** application on your ***Windows 95*** or ***NT*** terminal. Download the "a.hex" file, which resulted from the last step. To do so, write `DL -k` on the ***DINK*** monitor. On the terminal it will appear, "`Set to Keyboard Port`". Press *Transfer->Send Text File* on the communication terminal's menu. Find your "a.hex" file and select it. The file will be downloaded to the ***Excimer*** board.

5. Execute the program by writing "`go 70000`" on the terminal.

6. Observe the behavior of the on-board LED's. What happens if you decrease/increase the value of *parameter* in your FOR loop statement?

References:

[1] Motorola, Designing a Minimal PowerPC System, PowerPC Application Note: AN1769/D, 1998.

Suggested Code:

```
/* This program will blink the status and Error LEDs alternatively
ten times. After that, both LEDs will be shut down. 0xfffff will
cause a visible delay in a 300MHz PowerPC*/
```

```
Main ()
{
   unsigned long count;
   int loop;
   for (loop = 0 ; loop <= 10; loop++)
   {
      *(char *) (0x40200000) = 0x00; //turn on status
      *(char *) (0x40600000) = 0x08; //turn off error
      for(count = 0; count <= 0xfffff; count ++);
      *(char *) (0x40200000) = 0x08; //turn off status
      *(char *) (0x40600000) = 0x00; //turn on error
      for(count = 0; count <= 0x1fffff; count ++);
   }
   *(char *) (0x40600000) = 0x08;
}
```

Conclusions:

Students should be able to note that:

- The speed, which drives the ***PowerPC*** microprocessor, is very fast and thus a blinking effect might not be perceived.
- For different loop parameters, the LED will remain ON or OFF for a different time period.
- The LED's are configured as Common Anode (negative terminal connected together).

Troubleshooting:

If the student is not able to turn ON or OFF the LED check that:

- the address being written to is either 0x40600000 or 0x40200000.
- a suitable value for the time delay loop has been defined.
- the student has compiled, linked and downloaded the program correctly.

Experiment

5

LED Control from Keyboard

Problem Statement:

- This experiment requires the compilation, downloading, and execution of a C language program which blinks the Excimer Board's ERROR Light Emitting Diode (LEDs) the number of times specified by the user input. (Contributed by Noel Serrano and José I. Quiñones).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- use the DINK functions presented in experiment #3
- print to the DINK 32 interface
- capture single characters from the keyboard and echo them to the DINK 32 interface

Procedure:

1. Write a C program that will blink the on board LED's based on user input. The program should ask the user which LED he wants to blink and how many times.

   Hint: To create this program use the program you created in the previous experiment and the Useful DINK Functions.

References:

[1] Motorola, Designing a Minimal PowerPC System, PowerPC Application Note: AN1769/D, 1998.

Suggested Code:

```
#include <stdio.h>

#define  getchar dink_get_char
#define  putchar dink_write_char
#define  printf dink_printf

void blink_leds(int addr, int i);
unsigned long (*dink_get_char)() = (unsigned long (*)()) 0x1e4c4;
unsigned long (*dink_write_char)(char) = (unsigned long (*)(char)) 0x5eb4;
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6270;

main()
{
      int decimal_no;
      char LED;
      char number;
      do
      {
            printf ("\nSelect the LED you want to blink:\n");
            printf ("\tS - Press S for the Status LED\n");
            printf ("\tE - Press E for the Error LED\n");
            printf ("\tQ - Press Q to Quit\n");
            LED = getchar();   /* Read typed Character */
            if (LED == 'E' || LED == 'e')
            {
                  printf ("\nEnter the number of times (1-9) to blink the Er-
                           ror LED: ");
                  do{  /* is it a number??? */
                        number = getchar();
                  }while ( !((number >= '0') && (number <= '9')) );
                  putchar(number);   /* echo typed character */
                  decimal_no = number - 48;
                  blink_leds(0x40600000, decimal_no);
            }
            else if (LED == 'S' || LED == 's')
            {
                  printf ("\nEnter the number of times (1-9) to blink the
                           Status LED: ");
                  do{
                        number = getchar();
                  }while ( !((number >= '0') && (number <= '9')) );
                  putchar(number);
                  decimal_no = number -48;
                  blink_leds(0x40200000, decimal_no);
            }
      } while ( LED != 'Q' && LED != 'q' );     /*  X or x */
    return 0;
}
```

```
void blink_leds(int addr, int i)
{
      unsigned long count;
      int loop;
      for (loop = 0 ; loop < i; loop++)
      {
            *(char *) (addr) = 0x00;      //turn on error
            for(count = 0; count <= 0xfff00; count ++);
            *(char *) (addr) = 0x08;      //turn off error
            for(count = 0; count <= 0xfff00; count ++);
      }
      *(char *) (0x40600000) = 0x08;
}
```

Conclusions:

Students should be able to note that:

- a PowerPC Excimer Board program can obtain data from a user via the Keyboard.
- the getchar function is not useful in cases you need more than character as input, so an implementation of a scanf function would be useful.

Troubleshooting:

If the student is not able to:

- Access DINK 32 interface functions: use the `st` command to verify that the address for the DINK functions matches the ones provided in this manual.

- Blink the LEDs: verify the memory mapping for each of the LEDs.

Experiment

6

A simple scanf function for Excimer

Problem Statement:

- In this experiment the student will develop a C function for taking character input from the terminal emulator's keyboard attached to Excimer through the serial port and converting number characters to decimal values used in other programs. (Contributed by Chuck Corley, Motorola)

Objectives:

Upon completion of this laboratory experience, students will be able to:

- substitute the getchar and putchar equivalent functions available in DINK for the same functions normally found in <stdio.h>
- recognize the ASCII character values returned from getchar() and echo them back via putchar()
- convert digit characters input through the keyboard into decimal integer values for use in other programs
- utilize DINK's print output function to display the resulting decimal value

Background Information:

Texts on programming describe how to get input for a program. For example, The Waite Group's New C Primer Plus [1] says:

> The C library contains several input functions, and scanf() is the most general of them, for it can read a variety of formats. Of course, input for the keyboard is text because the keys generate text characters: letters, digits, and punctuation. When you desire to enter, say, the integer 2002, you type the characters 2 0 0 and 2. If you want to store that as a

numerical value rather than as a string, your program has to convert the string character-by-character to a numerical value. And that is what scanf() does! It converts a string input into various forms: integers floating-point numbers, characters, and C strings.

It is the inverse of printf(), which converts integers, floating-point numbers, characters, and C strings to text that is to be displayed on the screen. Like printf(), scanf() uses a control string followed by a list of arguments. The control string indicates into which formats the input is to be converted.

The DINK software on Excimer provides input and output functions that save the programmer from having to interact directly with the duart that receives input and sends output to the terminal. However these functions are not at the level of a complex function like scanf(). Nevertheless, many of the C programs that we desire to run on Excimer call the scanf() function because of it's widespread use.

In this experiment, you will write your own function my_scanf() and substitute it (by a #define directive) for any scanf() function that the compiler may encounter in programs intended for Excimer. Likewise you will define dink_printf() to substitute for printf() and link dink_printf() into your programs. Then you will have input and output functions for use in other programs.

To keep my_scanf() simple we will assume that the only control string for converting inputs is the %d or decimal format. Your my_scanf() function should accept a control string as an argument but then ignore it and return a decimal value to the second (and last) argument in the functional call. Later experiments may require more sophisticated substitute functions for scanf(), but this simple decimal input routine will be widely applicable.

Eximer's dink_printf() does accept a control string but it ignores floating-point and character formats. It will only print decimal numbers (%d), hexadecimal numbers (%x), and strings (%s) and then only one such format per printf statement.

Procedure:

1. Write a C language program which asks the user to input a number through the keyboard and then outputs the number input as a positive decimal number.
2. In a separate file write a C program my_scanf(char, int) which reads characters from the keyboard, echoes those that are digits, and at the carriage return assigns a decimal value to the second argument of my_scanf().

Hint: While ignoring non-digit characters may be an acceptable simplification, you may want to check for backspace or delete characters and take the appropriate action if the user attempts to correct his numerical input.

3. Write a header file which equates the function name scanf to my_scanf and printf to dink_printf. In the header file equate dink_printf to the address where it is stored in RAM as revealed by DINK's symtab (symbol table) command. Include this header file in your test program.

Example:

```
/*  File - support.h
 *  Equates functions used in Excimer Exercise to equivalent
 *  functions defined in DINK or in my_scanf.c
 *  NOTE:  If DINK function addresses change because DINK changes,
 *  addresses here must be changed accordingly.
 *
 *  Modification history:
 *  19Jan99,CJC Original
 */

#define  printf dink_printf
#define  scanf my_scanf

extern void my_scanf(const char *, ...);

unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6368;
```

4. Your my_scanf() function will use getchar() and putchar(). Write a header file equates these to DINK's get_char and write_char. Write a header file which equates dink_get_char() and dink_write_char() to the addresses where they are stored in RAM as revealed by DINK's symtab (symbol table) command. . Include this header file in the my_scanf.c program.

Example:

```
/*  File - excimer.h
 *  Provides the addresses of functions defined in DINK on the Excimer
 *  board and used by programs.  The addresses of the functions are
 *  taken from the xref.txt file generated by the linker.
 *  When a new version of DINK is downloaded to the target, make sure
 *  the functions' addresses are changed accordingly to match with the
```

```
 *  new addresses being generated.
 *
 *  Modification history:
 *  21Oct98,My Created for ExcDemo
 *  19Jan99,CJC Modified to run with my_scanf code.
 */

#define  getchar dink_get_char
#define  putchar dink_write_char

/* Addresses of DINK functions.  */
unsigned long (*dink_get_char)() = (unsigned long (*)()) 0x1e5e4;
unsigned long (*dink_write_char)(char) = (unsigned long (*)(char)) 0x5fac;
```

5. Link your input/output test program and my_scanf program.
6. Download the resulting S-record file to Excimer, execute it, confirm that it echoes only digit characters and returns the correct decimal value to your program at the carriage return.

References:

[1] The Waite Group's New C Primer Plus (1990: Howard W. Sams & Co, Carmel, IN)

Suggested Code:

```
/* file "testscanf.c"
 * A test harness for Excimer Experiment to prove out
 * my_scanf() function.
 * Modification History:
 * 990121  CJC  Original
*/

#include "support.h"

void main(void)
{
        int     decimal_no;
        printf ("Enter a decimal number: ");
        scanf("%d", &decimal_no);
        printf ("\nDecimal number is: %d \n", decimal_no);

/* file "support.c"
 * Defines an alternative to the scanf function provided by
 * stdio.h for use when running the Dhrystone benchmarks on DINK.
 * Created:  990119    CJC
 * Modified:
*/
#include "excimer.h"

void my_scanf(char *fmt, int *v)
{
        char    ch;
        int     no_runs = 0;

        while ((ch = getchar()) != 0xd)      /*  Carriage return? */
        {
                if ( (ch == 0x7f) || (ch == 0x8))  /*  Delete?  */
```

```
            {
                    putchar(0x8);         /* Backspace  */
                    putchar(0x20);        /* Overprint a space. */
                    putchar(0x8);         /* Backspace  */
        /* Assume modulo arithmetic to subtract last digit added. */
                    no_runs = no_runs / 10;
            } else
            if ( (ch >= '0') && (ch <= '9'))  /*  A digit?  */
            {
                    putchar(ch);          /* Echo it and  */
                    /* Accumulate the value.  */
                    no_runs = (no_runs * 10) + (ch - 48);
                    /* ASCII character - 48 equals the digit. */
            }
        }
        *v = no_runs;         /* Assign second Arg the value. */
}

/* file "makefile"  */
SUPPORT =
SUPPORTOPT  =
OPTLEV  = -O1
CPU          = 603
TARGFLAGS   = -Hppc$(CPU)

CC = c:\sw\metaware\hcppc\bin\hcppc -Ic:\sw\metaware\hcppc\inc \
        -Hnocopyr -c -nofsoft  $(OPTLEV) $(TARGFLAGS)
AS = c:\sw\metaware\hcppc\bin\asppc -c -big_si
LKOPT  =  -Bbase=0x70000 -xm -e main -Bnoheader -Bhardalign \
            -xo=$(@) -q -Qn -Cglobals -Csections -Csymbols -Ccrossref \
               > $(@D)\xref.txt
LINK    =  c:\sw\metaware\hcppc\bin\ldppc $(LKOPT)

testscanf.src: testscanf.o my_scanf.o
        $(LINK) testscanf.o my_scanf.o \
        c:\sw\metaware\hcppc\lib\be\fp\libmw.a

testscanf.o: testscanf.c
        $(CC) testscanf.c -o testscanf.o $(SUPPORTOPT)

my_scanf.o: my_scanf.c
        $(CC) my_scanf.c -o my_scanf.o $(SUPPORTOPT)
```

Conclusions:

Students should be able to note that:

- Characters are received from the keyboard as bytes of ASCII encoded information.
- Input/Output functions normally available in standard C libraries for a given computer may not be available or may exist in different, simpler forms on a small, embedded evaluation system like Excimer .
- Programmers can write their own input/output routines or link in routines that are provided in the embedded system.

- Hard-coding addresses of embedded routines is a dangerous way of linking code if the routines are relocated by DINK revisions.

Troubleshooting:

If the student is not able to:

- Get started. Suggest that the student develop and debug the C program on the host computer by including <stdio.h> before substituting the DINK routines and downloading to Excimer. This should clarify the ASCII encoding of digits and conversion to a decimal number.
- Recognize the carriage return character. Eximer will be in a continous loop of accepting and echoing input. Additional printf() statements which output each character as it is read in will reveal the value provided by the duart for the carriage return character.Troubleshooting:

Experiment

7

Introduction to Assembly Language Programming

Problem Statement:

- In this experiment the student is introduced to the PowerPC instruction set architecture through the development of an assemblylLanguage routine. (Contributed by Eisen Montalvo-Ruiz)

Objectives:

Upon completion of this laboratory experience, students will be able to:

- write and compile an assembly language subroutine
- use Metaware Assembler directives
- understand the instruction set and the register set of the PowerPC

Background Information:

- **PowerPC Register Set**

  The PowerPC architecture has two levels of privileges, the user mode, and the supervisor mode. In the supervisor mode all registers are available to the programmer, while in the user mode only a subset of the registers are available. We are going to focus on the user mode for this laboratory.

  Tin the user mode the available PowerPC registers include 32 General Purpose Registers (GPRs), 32 Floating-Point Registers (FPRs), a Condition Register (CR), a Floating-Point Status and Condition Register (FPSCR), the XER register, the Link Register (LR) and the Count Register

(CTR). In addition, there are two read-only registers, associated with the Time Base Facility (TBU and TBL).

The GPRs are used to manipulate integer data. They come in two sizes, according to the implementation of processor. 32-bit GPRs for the 32-bit PowerPC and 64-bits for the 64-bit PowerPC. They are used as source and destination registers in the integer instructions.

The FPRs are used with floating-point instructions. They are 64 bits wide independently of the implementation, and can manipulate single- and double floating-point data. Related to these registers is the FPSCR. It contains all floating-point exception signal bits, excluding summary bits, exception summary bits, exception enable bits, and rounding control bits.

The CR is a 32-bit register, divided into eight 4-bit fields. This register contains the results of certain arithmetic operations and provides a way for testing and branching. The XER register indicates overflows and carry conditions for integer operations. The LR register and the CTR register are like the GPRs, their size depends on the implementation. The LR supplies the branch target address for the Branch Conditional to Link Registers instructions. The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions.

The Time Base Facility consists of a 64-bit register, divided in two 32-bit registers, Time Base Upper (TBU) and Time Base Lower (TBL). These registers will be used in a future laboratory, where you will learn more about them.

- **PowerPC Instruction Set**

  The PowerPC Instruction Set is very powerful and extensive. It contains around 200 instructions, excluding suffices. We don't have the space to cover all of them. For now, we are going to work with the Integer Arithmetic, Load and Store, and Flow Control instructions. A general description

of the format of the instructions will be given. More information can be obtained from the PowerPC programming references.

Integer Instruction Set

(a) Integer Arithmetic Instructions

You can add, subtract, multiply, and divide integer numbers. You can use immediate values and registers. Also, register to register instructions are available. A general description of the format of the instructions follows.

1. Immediate Values

   opcode rD, rA, SIMM - where rD is the destination register, rA is the source register and SIMM is a Signed Immediate value.

2. Register to Register

   opcode rD, rA, rB - where rD is the destination register and rA and rB are the source registers.

(b) Integer Compare Instructions

These instructions can be used in conjunction with the branch instructions to control the flow of a program. They affect the CR, such that the branch instructions can choose their target address based on what happened in the previous instruction. Of course, they could be used only for comparing.

1. Immediate Values

   opcode rA, SIMM - where rA is the register you want to compare to a Signed Immediate value

2. Register to Register

   opcode rA, rB - where rA is the register you want to compare to register

rB

Load and Store Instruction Set

Load and Store instructions allow data movement between memory and register locations. They have three addressing modes. In anyone of them, if you use r0, the address calculation will use zero instead of the value in rA.

(a) Register Indirect with Immediate Index Addressing

`opcode rD, SIMM(rA)` - if loading then rD is the destination register. It will contain the value that is stored in the memory address that is the sum of SIMM and the value in the register rA. If storing then the memory address that is the sum of SIMM with the value in register rA, will contain the value stored in register rD.

(b) Register Indirect with Index Addressing

opcode rD, rA, rB - if loading then rD is the destination register. It will contain the value that is stored in the memory address that is the sum of the value in register rA and the value in the register rB. If storing then the memory address that is the sum of the value in register rA with the value in register rB, will contain the value stored in register rD.

(c) Register Indirect Addressing

opcode rD, rA - if loading then rD is the destination register. It will contain the value that is stored in the memory address that is the value in the register rA. If storing then the memory address that is the value in register rA will contain the value stored in register rD.

Branch Instructions Set

These instructions are commonly used with compare instructions. You place the branch after the compare, using the result of it to make the decision.

opcode label - where label is the address of the code where you want to branch to. The assembler takes care of translating the label to the address.

- **Metaware Assembler Directives**
  The assembler directives are instructions to the assembler on how to configure data and where to put the code and data in memory. The most useful are:
  (a) .text – identifies where the code section starts.
  (b) .data – mark the start of the data section
  (c) .word <value> – reserves space for a word in memory
  (d) .org <address> – starting address of the following code and/or data
  (e) .global <label> – makes this routine a public one.

  You can put comments in any line, but they must begin with a "!". In addition, you can use labels for branching. They must end with a semicolon and must be at the beginning of the line, with or without code in the same line.

- **Metaware Assembly Compilation**
  For compiling your code using Metaware, you must go through two steps. First compile the code using `asppc`, the Metaware Assembler.

```
asppc -o filename.o filename.s
```

  The extension of yo

ur file must be *.s. In this way the Assembler recognizes the file. The –o option tells the assembler the name of the object file. If you don't use it, the default name is the same as the code file with *.o as the extension.

The second step is to convert the object to Motorola S3 record and to set again the address of the code and data section.

```
elf2hex -p .text:0x70000,.data:0x70100 -o filename.hex -xm filename.o
```

The –p option is used to tell where the section of the file starts in memory. In this case, .text section will start in address 0x70000 and the .data section in 0x70100. The –o option is the name of the output file. The –xm tells the program to generate a Motorola S3 record and filename.o is the file of the object file.

Procedure:

1. Write an assembly language routine that multiplies 2 3x3 Matrices.

Remember:

$$\begin{matrix} a_{11} & a_{12} & \dots & a_{1j} \\ a_{21} & a_{22} & \dots & a_{2j} \\ \dots & \dots & \dots & \dots \\ a_{i1} & a_{i2} & \dots & a_{ij} \end{matrix} \leftrightarrow \begin{matrix} b_{11} & b_{12} & \dots & b_{1j} \\ b_{21} & b_{22} & \dots & b_{2j} \\ \dots & \dots & \dots & \dots \\ b_{i1} & b_{i2} & \dots & b_{ij} \end{matrix} = \begin{matrix} c_{11} & c_{12} & \dots & c_{1j} \\ c_{21} & c_{22} & \dots & c_{2j} \\ \dots & \dots & \dots & \dots \\ c_{i1} & c_{i2} & \dots & c_{ij} \end{matrix}$$

$C_{ij} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \dots + a_{in} * b_{nj}$, where n is the matrix dimension.

Hint:

Set the start of the matrices in memory, so you know where the code has to look for the data. Also, make it flexible, so you can change the size of the matrices without making changes in your code.

References:

[1] Motorola, PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, MPCFPE32B/AD, Rev 1, 1/97.

Suggested Code:

```
!file "matmult.s"
! Assembly Language program to multiply 2 3x3 matrices
! EMR 990321
!
! Register usage:
! r9  - Pointer to start of data section
! r12 - miscellaneous
! r5  - i
! r10 - j
! r11 - k
! r7  - Pointer to row in matrix
! r8  - Pointer to column in matrix
! r5  - holds temporary result of calculations

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! CODE Section                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        .org 0x60000
         .text
        .global mat_mult
mat_mult:
        addic r5, r0, 0       !Clear R5

        lis   r9, 6           !Load immediate shifted to R9
        addi  r9, r9, 208     !Pointer to data section

        lwz   r12, 108(r9)    !Load n

        cmpw  r5, r12         !If R5>=R12 then ...
        bge   exit            !goto exit, else ...

another_i:
        addi  r10, r0, 0      !Clear R10

        lis   r9, 6           !Load immediate shifted to R9
        addi  r9, r9, 208     !Pointer to data section

        lwz   r12, 108(r9)    !Load n

        cmpwi r12, 0          !If R12<=0 then ...
        ble   incr_i          !goto incr_i, else ...

another_j:
        addi  r11, r0, 0      !Clear R11

        lis   r9, 6           !Load immediate shifted to R9
        addi  r9, r9, 208     !Pointer to data section
```

```
        lwz   r12, 108(r9)    !Load n

        cmpwi r12, 0          !If R12<=0 then ...
        ble   incr_j          !goto incr_j, else ...

another_k:
        mulli r7, r5, 12      !Pointer to row of A using i
        slwi  r12, r11, 2     !Pointer to col of A using k

        add   r12, r12, r7    !Pointer to Aik

        lis   r9, 6           !Load immediate shifted to R9
        addi  r9, r9, 208     !Pointer to data section

        lwzx  r6, r12, r9     !Load Aik to R6

        mulli r12, r11, 12    !Pointer to row of B using k
        slwi  r8, r10, 2      !Pointer to col of B using j

        add   r12, r8, r12    !Pointer to Bkj

        add   r12, r12, r9    !Add start address of data section
        lwz   r12, 36(r12)    !Load Bkj

        mullw r6, r12, r6     !Aik*Bkj

        add   r12, r8, r7     !R12=i+j
        add   r8, r12, r9     !Pointer Cij
        lwz   r12, 72(r8)     !Load Cij
        add   r12, r12, r6    !Cij+=Aik*Bkj

        stw   r12, 72(r8)     !Store Cij

incr_k:
        addi  r11, r11, 1     !Increment k
        lwz   r12, 108(r9)    !Load n

        cmpw  r12, r11        !If R12>R11 then ...
        bgt   another_k       !goto another_k, else ...

incr_j:
        addi  r10, r10, 1     !Increment j

        lis   r9, 6           !Load immediate shifted to R9
        addi  r9, r9, 208     !Pointer to data section

        lwz   r12, 108(R9)    !Load n

        cmpw  r12, r10        !If R12>R10 then ...
        bgt   another_j       !goto another_j, else ...

incr_i:
        addi  r5, r5, 1       !Increment i

        lis   r9, 6           !Load immediate shifted to R9
        addi  r9, r9, 208     !Pointer to data section
```

```
        lwz   r12, 108(R9)    !Load n

        cmpw  r5, r12         !If R5<R12 then ...
        blt   another_i       !goto another_i, else ...
exit:
        blr                   !exit

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!DATA section                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        .org 0x600d0
         .data
matrix_a:
        .word 1
        .word 1
        .word 1
        .word 1
        .word 1
        .word 1
        .word 1
        .word 1
        .word 1
matrix_b:
        .word 2
        .word 2
        .word 2
        .word 2
        .word 2
        .word 2
        .word 2
        .word 2
        .word 2
matrix_c:
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
n:
        .word 3
```

Conclusions:

Students should be able to note that:

- Programming in assembly is a bit complex. However, the increase in performance and the smaller size of the resulting code makes it worth in some cases.
- This routine alone is not very useful, but in the next laboratory we are going to show a way to interface an assembly routine to a C/C++ program.

Troubleshooting:

If the student is not able to:

- Get started. Suggest that the student code the multiplication in a C program until they have proven their algorithm. If they are still having difficulty, the disassembly of the c program could provide insight.

Experiment

8

Linking Assembly Language and C Code

Problem Statement:

- This experiment introduces the student to linking PowerPC assembly language and C code. (Contributed by Eisen Montalvo-Ruiz)

Objectives:

Upon completion of this laboratory experience, students will be able to:

- call an assembly routine from a C/C++ program
- know the PowerPC function-calling sequence

Background Information:

The following information is excerpted directly from Chapters 10 and 11 of the "High C/C++ Programmer's Guide for PowerPC". This document can be obtained from Metware through their website.

- **Making an assembly routine callable from a C program**

To be able to call an assembly language routine from a C program you must insert this piece of code before the assembly routine:

```
        .text
        .align 2
        .global name
name:
```

You are going to use "name" to call the routine from a C program.

- **Calling an assembly routine from C**
  For each assembly function you want to call, you have to declare it external. Then use the pragma directive Alias for linking the internal name to the external name. The following code should make it clearer:

```
extern foobar();
#pragma Alias(foobar,"name");
...
void main()
{
        ...
        foobar();
        ...
}
```

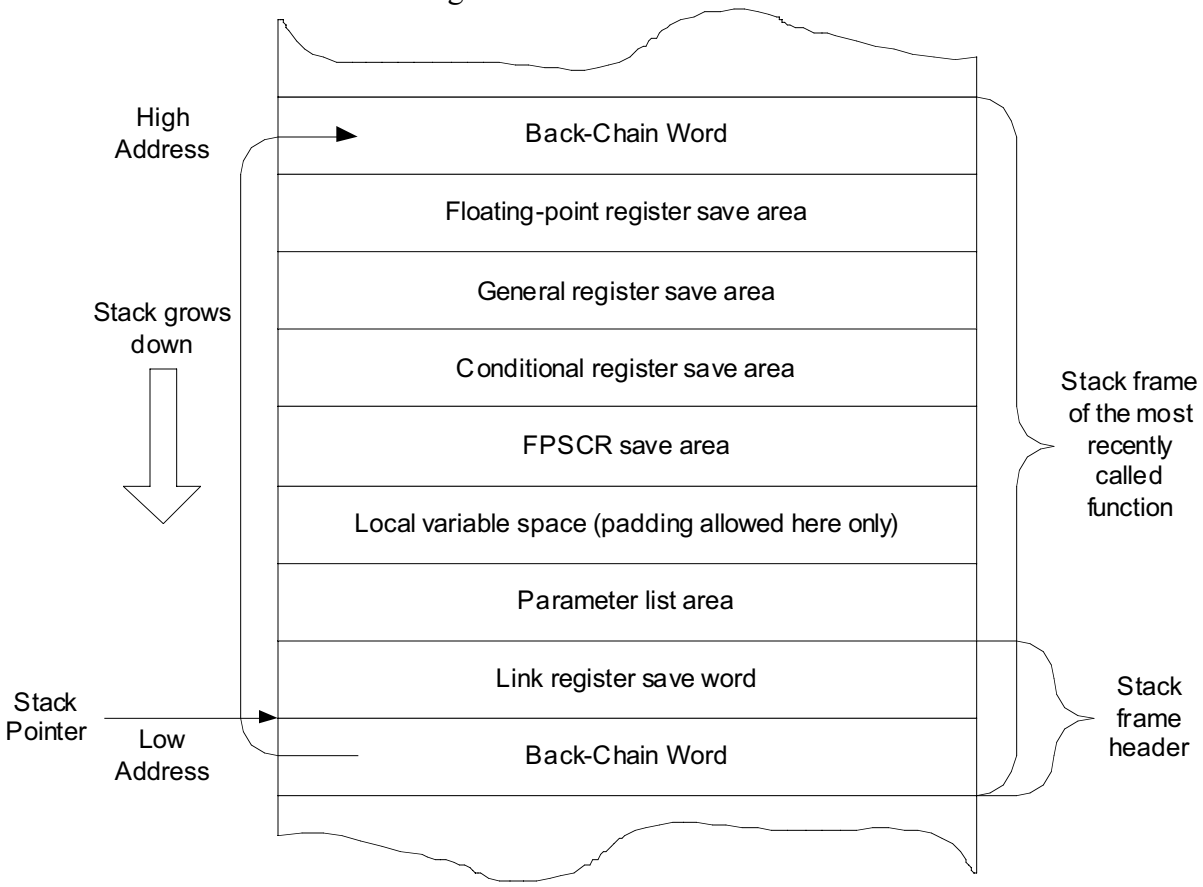
- **Function-Calling Sequence**

One of the most difficult parts of assembly language programming is parameter passing in function calls. Fortunately, the PowerPC function-calling and parameter passing is among the easiest one in the realm of assembly programming. Here goes a brief description of this process. If you want more information, read the books in the reference section.

Stack-Frame Layout

Figure 7.1 shows the memory stack frame organization for the PowerPC system. Every function needs to establish their stack frame, but the stack frame is only necessary if the function is going to call another function.

Figure 7.1 Standard Stack Frame



The stack frame grows downward from high to low memory address, and is 16-byte aligned. It doesn't have a maximum size but it has a minimum. The minimum stack frame consists of the stack-frame header, with padding to a 16-byte alignment. Any padding must occur within the local variable area. The Stack pointer points to the Back-Chain word of the most recently called function. This forms a linked list of stack frames.

The stack frame can include the following areas as required by any function:

- Floating-point register save area – non-volatile floating-point registers modified
- General register save area – non-volatile general registers modified
- CR save area – condition register fields modified
- FPSCR save area – floating-point status and control register bits modified
- Local variable space – local variables of function not mapped to registers

- Parameter list area – allocated by the caller of function; must be large enough to contain the arguments that the caller stores in it
- LR save word – contents of the link register as they were at the time of entry to a function
- Back-chain word – pointer to the previous stack frame's back-chain word

The parameter list area is not preserved across function calls and it must follow the stack frame header immediately.

Register usage

Table 7.1 contains the usage and status of the registers in the function calling process. Non-volatile registers "belong" to the calling function. If the called function wants to use them, it must save their values before using the registers and restore them before returning.

Volatile registers are not preserved across function calls, so you can use them without saving them. Also you can't use the dedicated and reserved registers. You can corrupt the system if you use them.

Table 7.1 PowerPC Register Usage

Register Name	Status	Usage
r0	Volatile	Language-specific purposes
r1	Dedicated	Stack frame pointer, always valid
r2	Dedicated	Reserved for system use
r3-r4	Volatile	Parameter passing and return values
r5-r10	Volatile	Parameter passing
r11-r12	Volatile	Language-specific purposes
r13	Reserved	Small data area pointer
r14-r30	Non-Volatile	Local variables
r31	Non-Volatile	Local variables or "environment pointer"
f0	Volatile	Language-specific purposes
f1	Volatile	Parameter passing and return values
f2-f8	Volatile	Parameter passing
f9-f13	Volatile	Scratch
f14-f31	Non-Volatile	Local variables

CR0 CR1 CR2 CR3 CR4 CR5 CR6 CR7	Volatile Volatile Non-Volatile Non-Volatile Non-Volatile Volatile Volatile Volatile	Condition Register fields, each four bits wide (Bit 6: Floating-point invalid operation exception)
LR	Volatile	Link Register
CTR	Volatile	Count Register
XER	Volatile	Fixed-Point Exception Register
FPSCR0-23 FPSCR24-31	Volatile Modifiable	Floating-Point Status and Control Register (Exception-enable and rounding-control bits)

Parameter passing

A maximum of eight integer arguments can be passed in general purpose registers r3 through r10 and a maximum of eight floating-point arguments can be passed in f1 through f8. If the number of parameters is less than the maximum, the unneeded registers contain undefined values. If the parameters passed do not fit in those registers, the function must allocate a stack frame. It should allocate the minimum space needed for the parameters that do not fit in the registers.

If the function wants to return a value, the table 7.2 shows how they can be passed, according to their type.

Table 7.2 PowerPC Function Return Values

Function Return Type	**Return in Register**	**Comment**
float	f1	
double		
int	r3	Returned as unsigned or signed integer (as appropriate), zero- or signed-extended to 32 bits if necessary
long		
enum		
short		
char		
pointer to any type		
long long	r3 and r4	Returned with the lower-addressed word in r3 and the higher-addressed word in r4
unsigned long		

struct(less than or equal to 8 bytes) union(less than or equal to 8 bytes)	r3 and r4	It is returned as if the following steps had occurred: 1- The struct or union was first stored in an 8-byte aligned memory area. 2- The low-addressed word was loaded into r3 3- The high-addressed word was loaded into r4
long double struct(greater than 8 bytes)	Storage Buffer	The address of this buffer is passed as a hidden argument in r3

- **Metaware Compiling**

When you are combining assembly and C, you can't compile like you did in the last laboratory. This is an example of compiling C and assembly using hcppc, the Metaware C/C++ compiler.

```
hcppc -Hppc603 -Hldopt=-e,main -Hldopt=-B,start_addr=70000
-Hldopt=-x -Hldopt=-m c_code.c assembly_code.s
```

The option –Hppc603 tells the compiler to generate PowerPC 603 code. The –Hldopt are options passed to the linker. The value after the equal sign is the option and the value after the comma is the value of the option. For example –e tells the linker what function is the starting point in the code. In this example, the starting point is the main function. The –B has a lot of values. One of the most useful is start_addr. It tells where the code starts in memory. In this example, the code starts at 0x70000. The –m generates a map list file of the code. The standard output is the screen. You can use redirection to send the output to a file. And finally the –x tells the linker to generate Motorola S3 records, ready to be downloaded to the Excimer Board. The last parameters are the filenames of the C and Assembly code. The output code will be named "a.hex".

Procedure:

1. Write an assembly language routine that multiplies two *N* x *N* matrices and a C language program that asks the user for the size of the matrices, their initial values and shows the resulting matrix. The C program should call the assembly routine.

Hint: You can use the assembly routine you made in the last laboratory. If you followed the hint in that laboratory, you shouldn't need to make too many changes.

References:

[1] Motorola, PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, MPCFPE32B/AD, Rev 1, 1/97.

Suggested Code:

```
/*    file MatrixMult.c
      C program that calls an assembly routine. It asks the user for
      the size and initial values for the matrices and then shows the results
      of their multiplication.

      EMR 990407
*/
#define  scanf my_scanf  /* Useful Functions */
#define  getchar dink_get_char
#define  putchar dink_write_char
#define  printf dink_printf

void my_scanf(char *, int *);
/* Pointers to functions in Dink Memory */
unsigned long (*dink_get_char)() = (unsigned long (*)()) 0x1e4c4;
unsigned long (*dink_write_char)(char) = (unsigned long (*)(char)) 0x5eb4;
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6270;

/* Assembly Routine */
extern matrixmult(int size, int *result, int *mata, int *matb);
/* Alias(internal name, external name)
#pragma Alias (matrixmult,"matmult")

int *mata; /* First Matrix */
int *matb; /* Second Matrix */
int *matc; /* Resultant Matrix */
int size; /* Matrices Size */

int *malloc(unsigned int); /* Memory Allocation function proto*/

void main()
{
      int i, l;
      int temp;

      /* Ask the user for the size */
      printf("Enter size of matrices > ");
      scanf("%d",&size);
      printf("\n");

      /* Separate memory for the matrices */
      mata = malloc(size*size);
      matb = malloc(size*size);
      matc = malloc(size*size);
```

```
/* Ask user for initial values */
for(int j=0; j<size; j++)
{
	for(int m=0; m<size; m++)
	{
		printf("A%d",j+1);
		printf("%d = ",m+1);
		scanf("%d", &temp);
		printf("\n");
		mata[ j*size+m] =temp;

		printf("B%d",j+1);
		printf("%d = ",m+1);
		scanf("%d", &temp);
		printf("\n");
		matb[ j*size+m] =temp;

		/* Clear resultant matrix memory */
		matc[ j*size+m] =0;
	}
}

/* Calling assembly routine */
matrixmult(size, matc, mata, matb);


/* Display results */
for(i=0; i<size; i++)
{
	printf("| ");
	for(l=0; l<size; l++)
		printf("%d ",mata[ i*size+l] );
	printf("| ");
	if((i+1)==(size/2))
	{
		printf("*");
	}
	else
	{
		printf(" ");
	}
	printf(" | ");
	for(l=0; l<size; l++)
		printf("%d ",matb[ i*size+l] );
	printf("| ");
	if((i+1)==(size/2))
	{
		printf("=");
	}
	else
	{
		printf(" ");
	}
	printf(" | ");
	for(l=0; l<size; l++)
		printf("%d ",matc[ i*size+l] );
	printf("|");
	printf("\n");
```

```
      }
}

/* User Input Function */
/* By Chuck Corley     */
void my_scanf(char *fmt, int *v)
{
      char  ch;
      int   no_runs = 0;

      while ((ch = getchar()) != 0xd)     /*  Carriage return? */
      {
            if ( (ch == 0x7f) || (ch == 0x8))  /*  Delete?  */
            {
                  putchar(0x8);           /* Backspace  */
                  putchar(0x20);          /* Overprint a space. */
                  putchar(0x8);           /* Backspace  */
      /* Assume modulo arithmetic to subtract last digit added. */
                  no_runs = no_runs / 10;
            } else
            if ( (ch >= '0') && (ch <= '9'))  /*  A digit?  */
            {
                  putchar(ch);            /* Echo it and  */
                  /* Accumulate the value.  */
                  no_runs = (no_runs * 10) + (ch - 48);
                  /* ASCII character - 48 equals the digit. */
            }
      }
      *v = no_runs;           /* Assign second Arg the value. */
}

/* Memory Allocation Function */
/* By Chuck Corley            */
int *malloc(unsigned int size)
{
  static int buffer[2048];
  static int *next = buffer;
  int *p = next;
  next += ((size + 7) & ~7);
  if (next >= buffer + sizeof(buffer))
    /* Terminate by executing a zero.  */
    asm(".long 0");
  return p;
}

!file "matmult.s"
! Assembly Language program to multiply 2 3x3 matrices
! EMR 990407
!
! Parameters:
!     r3 = size
!     r4 = pointer to matrix c
!     r5 = pointer to matrix a
!     r6 = pointer to matrix b
!
! Register usage:
!     r14 = i
!     r15 = j
```

```
!     r16 = k
!     r17 = temp
!     r18 = offset to current value of cell in matrix a
!     r19 = offset to current value of cell in matrix b
!     r20 = offset to current value of cell in matrix c


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! CODE Section                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      .org 0x60000
      .text
      .align 2
      .global matmult
matmult:
      xor   r14, r14, r14       !Clear R14
      xor   r17, r17, r17       !Clear R23

      cmpw  r14, r3             !If i>=size then ...
      bge   exit                !goto exit, else ...

another_i:
      xor   r15, r15, r15       !Clear j

another_j:
      xor   r16, r16, r16       !Clear k

another_k:
      mullw r18, r14, r3        !Offset to row of A using i
      add   r18, r18, r16       !Offset to col of A using k
      slwi  r18, r18, 2         !Multiply by 4, we're loading words, not bytes
      lwzx  r18, r5, r18        !Load Aik to R21

      mullw r19, r16, r3        !Offset to row of B using k
      add   r19, r19, r15       !Offset to col of B using j
      slwi  r19, r19, 2         !Multiply by 4, we're loading words, not bytes
      lwzx  r19, r6, r19        !Load Bkj to R22

      mullw r17, r21, r22       !Aik*Bkj
      add   r23, r23, r17       !Cij+=Aik*Bkj

incr_k:
      addi  r16, r16, 1         !Increment k

      cmpw  r16, r3             !If k<size then ...
      blt   another_k           !goto another_k, else ...

save_val:
      mullw r20, r14, r3        !Offset to row of C using i
      add   r20, r20, r15       !Offset to col of C using j
      slwi  r20, r20, 2         !Multiply by 4, we're using words, not bytes
      stwx  r17, r4, r20        !Store Cij
      xor r17, r17, r17         !Clear Temp for another cell

incr_j:
      addi  r15, r15, 1         !Increment j

      cmpw  r15, r3             !If j<size then ...
```

```
      blt   another_j           !goto another_j, else ...

incr_i:
      addi  r14, r14, 1         !Increment i

      cmpw  r14, r3             !If i<size then ...
      blt   another_i           !goto another_i, else ...
exit:
      blr                       !exit
```

Conclusions:

Students should be able to note that:

- Linking assembly and C routines is very important in those cases where the complexity of the problem at hand requires a high level language but where the performance of certain routines within the problem is crucial.

Troubleshooting:

If the student is not able to:

- Get the parameters in the assembly function: use the list file to know where the assembly function and the parameters are in memory. Then look in the assembly code of the C program for the call to the assembly function. Before the call you will see where the code is putting the parameters in the registers for the assembly function. This could help you understand the function-calling sequence.

Experiment

9

Converting Integers to Floating Point

Problem Statement:

- This experiment requires the development of an assembly language subroutine to convert the 64 bit integer value read from the PowerPC time base facility to a 64 bit (double) floating point number representing seconds. (Contributed by Chuck Corley, Motorola)

Objectives:

Upon completion of this laboratory experience, students will be able to:

- write and assemble an assembly language subroutine
- call the assembly language subroutine from a C program and use the values returned
- convert integer numbers to PowerPC floating point representation
- convert time base count values to seconds of wall clock time

Background Information:

The PowerPC architecture requires each microprocessor implementation to provide a time base facility (TB), a 64-bit structure that consists of two 32-bit registers – time base upper (TBU) and time base lower (TBL). User level applications are permitted read-only access to the TB which is useful for timing program execution or providing a time reference. The update frequency of the time base is system-dependent so the algorithm for converting the current value in the time base to time of day is also system-dependent. The MPC603e microprocessor used on the Excimer board increments the TB at one-fourth the SYSCLK (bus) frequency.

Excimer does not have a real time clock chip as would be found on most computers. TBU and TBL are cleared at each power-up (or they can be set to an initial value in supervisor mode). The TB facility

then counts up at one-fourth of SYSCLK frequency from this initial value. Excimer cannot relate a TB value to real time without user assistance – like setting a watch.

SYSCLK is crystal controlled to 66.6666MHz (see the oscillator on the board at U15), therefore TBL increments 16,666,667 times per second. When TBL exceeds 2^{32}, a carry-out bit increments TBU. Thus, TBU will increment every 257.7 seconds and the total range of the TB is $1.1x10^{12}$ seconds or approximately 35,000 years. This number is better represented in application programs as a floating point value.

The PowerPC architecture represents double precision floating point values in the 64-bit format shown in Figure 1.

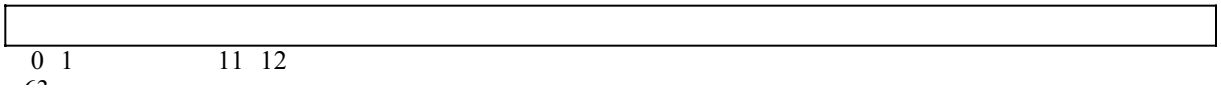


Figure 1. Floating-Point Double-Precision Format

Where

- S (sign bit)
- EXP (exponent + bias)
- FRACTION (fraction)

For numeric values, the significand consists of a leading implied bit concatenated on the right with the FRACTION. For normalized numbers (it is unnecessary to deal with denormalized floating point numbers in this excercise) the implied bit is a one and is the first bit to the left of the binary point. Normalized numbers are interpreted as follows:

$$NORM = (-1)^S \times 2^{(EXP - 1023)} \times (1.FRACTION)$$

The range covered by the magnitude (M) of a normalized double-precision floating-point number is approximately:

$$2.2x10^{-308} \leq M \leq 1.8 \times 10^{308}$$

The double precision exponent is biased by adding 1023 so that positive and negative exponents can be represented without a sign bit for the exponent. Example exponents are shown in Table 1.

Biased Exponent (Binary)	Unbiased Exponent (Double-Precision)
111_1111_1111	Reserved for infinities and NaNs
111_1111_1110	+1023
111_1111_1101	+1022
.	.
100_0000_0000	1
011_1111_1111	0
111_1111_1110	-1
.	.
000_0000_0001	-1022
000_0000_0000	Reserved for zeros and denormalized numbers

Table 1. Biased Exponent Format

Examples of TB integer values converted to double precision floating point representation are shown in Table 2. Since it would take 35,000 years to test a conversion program for the upper limits of the TB, this experiment should include a test program that supplies these example values to an integer-to-floating point assembly-language conversion routine and verifies that the correct floating point value is returned. The last column of Table 2 shows the floating point value of the TB converted to seconds given Excimer's 66MHz bus clock.

TB count (decimal)	TBU (hex)	TBL (hex)	S	EXP biased (dec)	FRACTION (hex)	DP Floating Pt Value (hex)	Seconds (dec)
0	0000_0000	0000_0000	0	0	0_0000_0000_0000	0000_0000_0000_0000	0.00
1	0000_0000	0000_0001	0	+1023	0_0000_0000_0000	3FF0_0000_0000_0000	6.00e-8
2	0000_0000	0000_0002	0	+1024	0_0000_0000_0000	4000_0000_0000_0000	1.20e-7
524,288	0000_0000	0008_0000	0	+1042	0_0000_0000_0000	4120_0000_0000_0000	3.15e-2
1.57e6	0000_0000	0018_0000	0	+1043	8_0000_0000_0000	4138_0000_0000_0000	9.44e-2
3.67e6	0000_0000	0038_0001	0	+1044	C_0000_8000_0000	414C_0000_8000_0000	2.20e-1

1.67e7	0000_0000	00FE_502B	0	+1046	F_CA05_6000_0000	416F_CA05_6000_0000	1.00
3.22e9	0000_0000	C000_0401	0	+1054	8_0000_8020_0000	41E8_0000_8020_0000	1.93e2
1.29e10	0000_0003	4000_5001	0	+1056	A_0002_8008_0000	4208_0002_8008_0000	7.73e2
1.58e16	0038_0001	4001_0005	0	+1076	C_0000_A000_8002	434C_0000_A000_8002	9.46e8
1.84e19	FEDC_BA98	7654_3210	0	+1086	F_DB97_530E_CA86	43EF_DB97_530E_CA86	1.10e12

Table 2. Example TB to Floating Point Conversions

Procedure:

1. Write an assembly language routine which accepts two unsigned integer arguments TBU and TBL and returns a double float value.

Suggestion: Assembly language routines are used primarily for speed (or access to hardware resources that are otherwise not available). To make this routine faster, try using static branch prediction. For example, a TB value of zero has to be tested as a special case to form EXP but is unlikely. Likewise, values over 2^{52} are unlikely (why would there be a conditional branch for this value?)

Hint: You may find the assembly language instructions cntlzw and rlwnm very useful.

2. Write a C program which calls the assembly language routine with the example values of Table 2 and check that it returns the correct floating point value.

Reminder: DINK Version 10.5 provides a `dink_printf` routine that may be used to print results to the terminal. However, it will not format floating point numbers; results will have to be displayed as two `unsigned long int` values. Is there a C construct which will permit viewing two 32-bit memory locations as both `unsigned long int` and `double`?

3. Write an assembly language routine that reads Excimer's TB facility and, using Excimer's bus clock speed of 66.6666Mhz, returns seconds as a double-precision floating point number.

Caution: TB must be read in two separate instructions. It is unlikely, but possible, that TBU could increment between reading these two registers. Consider the sequence TBU = 0x0000_0000, TBL = 0xFFFF_FFFF; TBU = 0x0000_0001, TBL = 0x0000_0000. What would be the error if your assembly language routine got the first value of TBU and the second value of TBL? Would reading the registers in reverse order avoid this problem?

Suggestions: This assembly language routine may be useful in other programs. Saving it in a standalone file "timer.s" and then linking it with this or other C programs will make it more useful. A header file, e.g. "Excimer.h," might be a convenient place to define constants like EXCIMER_BUS_SPEED that could change on other PowerPC systems.

4. Write a C program which outputs a zero to twenty second count to the terminal emulator and time it with a stopwatch. (Using `dink_printf` to display seconds as integer decimal numbers is acceptable).

References:

[1] Motorola, PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, MPCFPE32B/AD, Rev 1, 1/97.

Suggested Code:

```
/* file "Excimer.h"  */
/* Header file for Excimer-unique constants */

/* Excimer oscillator (U15 on PWB) runs at 66.6666MHz */
#define BUS_FREQUENCY 66666667

/* TB ticks/sec at Excimer bus clock. */
double  TICS_PER_SEC = BUS_FREQUENCY/4;

/* Bus frequency as an integer in MHz. */
int     IBUS_MHz = BUS_FREQUENCY/1000000;

/*  Excimer does not support <stdio.h>.  DINK has it's own print routine
 *  called dink_print which supports a limited number of format types (decimal-%d,
 *  hex-%x, string-%s, etc).  Redefining printf to point to the dink_print
 *  function, enables standard C programs downloaded to Excimer to print to the
 *  terminal.  The address of dink_print is available from the symbol table command
 *  (symtab) in DINK.  If the version of DINK on Excimer is updated from the Motorola
 *  website at http:\\www.mot.com\PowerPC\teksupport the address of dink_print must
 *  be updated here.
 */

#define  printf dink_print
unsigned long (*dink_print)() = (unsigned long (*) ()) 0x6368;


/* file "Exercise.h"
 *  Header file for common typedefs for Exercise? Chuck Corley  981218
*/

struct TB_View{   unsigned long TBU_View;
                  unsigned long TBL_View;
           };

union   DPFP_View  {   struct  TB_View TB_FPasGPR_View;
                  double  TB_FP_View;
           };

struct  Test_struct   {       struct TB_View TB_GPR_View;
                  union   DPFP_View     TB_FP_test;
```

```
        };


!file "dtime.s"  (For Metware High C/C++ Compiler/Assembler)
!  Assembly language routine to convert 64-bit PowerPC TB facility to
!  Double-precision, floating-point number.  (Plus additional routines for
!  testing.)   CJC 981216
!   Register usage:
!         r3 = FPU       (upper 32 bits of floating point value)
!         r4 = FPL       (lower 32 bits of floating point value)
!         r5 = TBU(time base upper - read from spr or loaded for test)
!         r6 = TBL(time base lower - read from spr or loaded for test)
!         r7 = leading zeroes in a register or shift count of +/-(zeroes - 11)
!         r8 = accumulator for final EXPonent value of DPFP number
!         r9 = shift count of 32 - n where n = +/-(zeroes -11)
!         r10 = constant register of 11
!         r11 = link register storage

#define TBU 269;         !Special purpose register numbers for TB
#define TBL 268;
    .data
Local_storage:
    .double 0
    .text
    .global dtime
    .global get_HID1
    .global conversion_test
!For CodeWarrior:
!asm double conversion(double TICS)
conversion:
    cntlzw  r7,r5          !Find leading zeroes in TBU. Preserve in r7.
    addi    r9,r0,32          !Will need a 32 in several places. Create one in r9.
    addi    r10,r0,11      !Create a constant in r10 = 11.
    subf.   r8,r7,r9          !r8 will hold EXP.  Currently (32 - leading zeroes)
    beq+    tbu_is_zero    !TBU never got incremented? (Zeroes=32?) (Most likely)

    subf.   r7,r10,r7      !No. Is TB more than 2^^52?  (Zeroes<11?) r7 = (Z-11)
    add     r8,r8,r9          !Final exponent will be (64 - 1 - leading zeroes).
    bge+    tbu_lt_8yrs    !If TB>2^^52, shift TBU bits right. (Not likely)

tbu_gt_8yrs:               !for Z<11: fpu = tbu>>n=(11-Z);
                                    !fpl = tbu<<n=(32-(11- Z))|tbl>>n=(11-Z);
    neg     r7,r7             !rlwnm shift count of (11-Z) = -(Z-11) = n = r7.
    subf    r9,r7,r9          !rlwnm shift count of 32-n = 32 - (11-Z) = r9.
    rlwnm   r3,r5,r9,12,31 !Shift TBU right n = (11 - Z). Mask off [ 0:11] .
    rlwnm   r4,r5,r9,0,10  !Shift remaining TBU bits left n = 32-(11-Z)
    rlwnm   r6,r6,r9,0,31  !Shift TBL right n = (11 - Z)
    or             r4,r6,r4          !Or rest of TBU shifted left with TBL shifted right.
    b       form_exponent  !Go bias the exponent and or into FPU.

tbu_lt_8yrs:               !for Z>=11: fpu=tbu<<n=(Z-11)|tbl>>n=(32-(Z-11));
                                    !fpl=tbl<<n=(Z- 11);
    subf    r9,r7,r9       !Form a shift count of 32 - (11-Z) = r9.
    rlwnm   r3,r5,r7,12,31 !Shift TBU left n = (Z-11). Mask off [ 0:11] .
    srw     r5,r6,r9       !Shift TBL bits right n = 32-(Z-11).
    or             r3,r3,r5   !Or TBU shifted left with TBL shifted right.
    rlwnm   r6,r6,r7,0,31  !Shift remainder of TBL left n = (Z-11).
    xor     r4,r6,r5       !XORing with the same value shifted right is like ANDing
    b       form_exponent  !fpl with a mask of all zeroes in bits [ 32-(Z-11):31] .

tbu_is_zero:               !Z= 32
    cntlzw  r7,r6          !Find leading zeroes in TBL.
    subf.   r8,r7,r9          !EXP = (32 - leading zeroes).
```

```
    beq-    tbl_is_zero    !Entire TBL count exactly zero?  (Not likely)
    subf.   r7,r10,r7      !No. Is TB less than 2^^20?  (zeroes < 11?)
    bge-    tbl_lt_63ms    !If not, will have to shift bits right. (Most likely)

tbl_gt_63ms:               !for z<11: fpu = tbl>>n=(11-z); fpl = tbl<<n=(32-(11-z));
    neg           r7,r7       !rlwnm shift count of (11-Z) = -(Z-11) = n = r7.
    subf    r9,r7,r9          !rlwnm shift count of 32-n = 32 - (11-Z) = r9.
    rlwnm   r3,r6,r9,12,31 !Shift TBL right n = (11 - z). Mask off [ 0:11] .
    rlwnm   r4,r6,r9,0,10  !Shift remaining TBL bits left n = 32 - (11 - Z).
    b       form_exponent

tbl_lt_63ms:               !for z>=11: fpu = tbl<<(z-11); fpl = 0;
    rlwnm   r3,r6,r7,12,31 !Shift TBL left n = (Z-11). Mask off bits 0-11.
    xor     r4,r4,r4          !fpl = 0.
    b       form_exponent

tbl_is_zero:               !for Z=32 && z=32: fpu = fpl = 0;
    xor     r3,r3,r3          !Unlikely result that TB was zero. Prepare to
    xor     r4,r4,r4          !return all zeroes for the floating point value.
    b compute_seconds

form_exponent:
    addi    r8,r8,1022     !Add DP bias (1023) -1 to the exponent
    rlwinm r8,r8,20,1,12  !Biased DP EXP will be (63-(leading zeroes in TB)+1023).
    or      r3,r3,r8

compute_seconds:
    lis     r5, Local_storage@h
    ori     r5, r5, Local_storage@l
    stw     r3, 0(r5)
    stw     r4, 4(r5)
    lfd     f2, 0(r5)      !Load back in as 64bit float
    fdiv    f1,f2,f1          !Divide by bus clock ticks per second
    blr                              !Return time in seconds as double in fp1

!  Routine passed sample values of TBU and TBL.  Returns FPU and FPL as
!  unsigned long.
!For CodeWarrior:
!asm struct TB_View * conversion_test(unsigned long Upper,
!                                    unsigned long Lower, double TICS)
conversion_test:
    or             r5,r3,r3          !Use test values of TBU and TBL passed in r3 and r4
    or             r6,r4,r4          !as substitutes for values read from TB.
    mflr    r11               !Save the return address.
    bl             conversion !Convert TBU and TBL into FPU and FPL
    mtlr    r11               !Return in r3 and r4
!For CodeWarrior:
!   la             r3,Local_pointer(SP)!Return a pointer to the FPU storage location.
    blr

!  Routine passed sample values of TBU and TBL.  Returns seconds as double.
!For CodeWarrior:
!asm double float_test(unsigned long Upper, unsigned long Lower, double TICS)
float_test:

    or             r5,r3,r3          !Use test values of TBU and TBL passed in r3 and r4
    or             r6,r4,r4          !as substitutes for values read from TB.
    mflr    r11               !Save the return address.
    bl             conversion !Convert TBU and TBL into FPU and FPL
    mtlr    r11               !Return as double in fpr1
    blr

! Routine reads the TBU and TBL. Returns seconds as double.
!For CodeWarrior:
```

```
!asm double dtime(double TICS)
dtime:
read_TB:
    mfspr   r5,TBU          !Get TBU.
    mfspr   r6,TBL          !Get TBL.
    mfspr   r7,TBU          !Get TBU again.
    subf.   r7,r5,r7            !Did it increment between reading TBU and TBL?
    bgt-    read_TB         !If so, read them again.  (Not likely)
    mflr    r11                 !Save the return address.
    bl      conversion      !Convert TBU and TBL into FPU and FPL
    mtlr    r11
    blr

! Routine reads the HID1 (PLL_CFG) register. Returns in r3.
get_HID1:
    mfspr r3,1009           !Get HID1 register.
    blr

/* file "test_program.c"
 *  Tests the operation of assy language routine to convert PowerPC TimeBase
 *  values from integer to DP floating point values.    Chuck Corley  981214
*/

#include <stdlib.h>
#include "Excimer.h"  /* File of Excimer board-specific constants */
#include "Exercise.h" /* File of common typedefs for this exercise */
struct TB_View conversion_test(int, int, double);  /* Given bus freq, returns time in seconds. */

void main(void)
    {
    int i, MAX_EXAMPLES;
    struct Test_struct       Example[ ] =
    {
/* Consider - Case1: Z<11; Case2: Z>=11; Case3: z<11; Case4: z>=11; Case5: Z=z=32; */
/* Case5: All leading zeroes. */
            {       0x00000000, 0x00000000, 0x00000000, 0x00000000},
/* Case4: Single one to treat as implied bit. Move from TB[ 32+31]  to DPFP[ 11] . */
            {       0x00000000, 0x00000001, 0x3FF00000, 0x00000000},
/* Case4: Single one to treat as implied bit. Move from TB[ 32+30]  to DPFP[ 11] . */
            {       0x00000000, 0x00000002, 0x40000000, 0x00000000},
/* Case4: Single one to treat as implied bit. Move from TB[ 32+12]  to DPFP[ 11] . */
            {       0x00000000, 0x00080000, 0x41200000, 0x00000000},
/* Case4: FRACTION starting in TB[ 32+12] . Move to DPFP[ 12:31] . */
            {       0x00000000, 0x00180000, 0x41380000, 0x00000000},
/* Case3: FRACTION starting in TB[ 32+11] . Move to DPFP[ 12:32] . Check DPFP[ 32] =1. */
            {       0x00000000, 0x00380001, 0x414C0000, 0x80000000},
/* Case3: FRACTION (One sec) starts TB[ 32+9] . Move to DPFP[ 12:34] . DPFP[ 32:34] =6? */
            {       0x00000000, 0x00FE502B, 0x416FCA05, 0x60000000},
/* Case3: FRACTION in TB[ 33:63] . FPU[ 12:31] =TBL[ 1:20] . FPL[ 0:10] =TBL[ 21:31] . */
            {       0x00000000, 0xC0000401, 0x41E80000, 0x80200000},
/* Case2: FRACTION-TB[ 31:63] . FPU[ 12] =TBU[ 31] . FPU[ 13:31] =TBL[ 0:18] . FPL[ 0:12] =TBL[ 19:31] .*/
            {       0x00000003, 0x40005001, 0x420A0002, 0x80080000},
/* Case1: FRACTION-TB[ 11:63] . FPU[ 12:31] =TBU[ 11:30] . FPU[ 0] =TBU[ 31] . FPL[ 1:31] =TBL[ 0:30] .*/
            {       0x00380001, 0x40010005, 0x434C0000, 0xA0008002},
/* Case1: TB[ 1:63] . FPU[ 12:31] =TBU[ 1:20] . FPL[ 0:10] =TBU[ 21:31] . FPL[ 11:31] =TBL[ 0:20] .*/
            {       0xFEDCBA98, 0x76543210, 0x43EFDB97, 0x530ECA86},
    };
    struct Test_struct Result;
    MAX_EXAMPLES = sizeof(Example) / sizeof(Example[ 0] );
    for (i=0; i< MAX_EXAMPLES; i++)
    {
/* These printf formats are for the restrictive dink_print routine. */
            printf("TBU= %x ", Example[ i] .TB_GPR_View.TBU_View);
            printf("TBL= %x ", Example[ i] .TB_GPR_View.TBL_View);
```

```
            Result.TB_FP_test.TB_FPasGPR_View = conversion_test (Example[ i] .TB_GPR_View.TBU_View, \
                                Example[ i] .TB_GPR_View.TBL_View, TICS_PER_SEC);

            if ((Result.TB_FP_test.TB_FPasGPR_View.TBU_View != \
                      Example[ i] .TB_FP_test.TB_FPasGPR_View.TBU_View) || \
                   (Result.TB_FP_test.TB_FPasGPR_View.TBL_View != \
                      Example[ i] .TB_FP_test.TB_FPasGPR_View.TBL_View))   \
                                    printf(" ERROR!\n");

            printf("FPU=  %x ", Result.TB_FP_test.TB_FPasGPR_View.TBU_View);
            printf("FPL=  %x\n", Result.TB_FP_test.TB_FPasGPR_View.TBL_View);

/*  This is not useful on Excimer because we can't print the floating point result.  It is a useful
check in CodeWarrior on the Mac.  CJC*/
/*
    Result.TB_FP_test.TB_FP_View = float_test(Example[ i] .TB_GPR_View.TBU_View,
                  Example[ i] .TB_GPR_View.TBL_View, TICS_PER_SEC);

            printf("FPR = %4.2e \n", Result.TB_FP_test.TB_FP_View);
*/
    };
    return;
    }

/* file "watch.c"
 *  Reads the PowerPC Time Base Facility on Excimer and prints out a twenty
 *  second count to the terminal emulator.    Chuck Corley  981214
 */

#include <stdlib.h>
#include "Excimer.h"  /* File of Excimer board-specific constants */
#include "Exercise.h" /* File of common typedefs */
double dtime(double); /* Given bus freq, returns time in seconds. */
unsigned long get_HID1();  /* Returns HID1 register. */
void main(void)
{
    double begin_time, current_time, delta_time = 0.0, seconds = 0.0;
    int int_seconds;
    unsigned long HID1_Reg;
    printf("PowerPC Timer Test.\n");
    printf("Beginning a twenty second count assuming bus speed of 66.67MHz.\n");
    printf("Please time me.\n");
    printf("If your stopwatch time differs significantly from 20 seconds, \n");
    printf("we can compute the actual bus speed.\n");
    begin_time = dtime(TICS_PER_SEC);
    for (int_seconds = -1; int_seconds <= 20; int_seconds++) /*Countup to start.*/
    {
        while (delta_time < 1.0)
        {
            current_time = dtime(TICS_PER_SEC);
            delta_time = current_time - (int_seconds) - begin_time;
        }
        delta_time = 0.0;
        switch (int_seconds)
        {
            case -1 :
                break;         /* Delay to get stopwatch ready. */
            case 0 :
                printf("Start now!\n");  /* Begin timing at zero seconds. */
                break;
            case 1:
                printf("%d second\n", int_seconds);
                break;
```

```
            default:
                printf("%d seconds\n", int_seconds);
        }   /* End of int_seconds switch  */
    };
    printf("If your time was not 20 seconds,\n");
    printf("bus speed is (20 / your_time) * 66.67MHz.\n");

/* Bonus Exercise.  Given the bus speed, calculate the processor (core) speed.*/
    HID1_Reg = get_HID1() >> 28;     /* Move HID1[ 0:3]  to [ 28:31] */
    printf("HID1 indicates PLL_CFG=%x.\n", HID1_Reg );
    printf("If bus=%dMHz, ", IBUS_MHz);
    switch (HID1_Reg)
    {
        case 0x4:  /*  PLL_CFG = 0b0100  */
            printf("Core Freq(2x)=%dMHz & ", 2*IBUS_MHz);
            printf("VCO Freq(2x)=%dMHz\n", 2*IBUS_MHz);
            break;
        case 0x5:  /*  PLL_CFG = 0b0101  */
            printf("Core Freq(2x)=%dMHz & ", 2*IBUS_MHz);
            printf("VCO Freq(4x)=%dMHz\n", 4*IBUS_MHz);
            break;
        case 0x6:  /*  PLL_CFG = 0b0110  */
            printf("Core Freq(2.5x)=%dMHz & ", (int)(2.5*(float)IBUS_MHz));
            printf("VCO Freq(2x)=%dMHz\n", 5*IBUS_MHz);
            break;
        case 0x8:  /*  PLL_CFG = 0b1000  */
            printf("Core Freq(3x)=%dMHz & ", 3*IBUS_MHz);
            printf("VCO Freq(2x)=%dMHz\n", 6*IBUS_MHz);
            break;
        case 0xe:  /*  PLL_CFG = 0b1110  */
            printf("Core Freq(3.5x)=%dMHz & ",(int)(3.5*(float)IBUS_MHz));
            printf("VCO Freq(2x)=%dMHz\n", 7*IBUS_MHz);
            break;
        case 0xa:  /*  PLL_CFG = 0b1010  */
            printf("Core Freq(4x)=%dMHz & ", 4*IBUS_MHz);
            printf("VCO Freq(2x)=%dMHz\n", 8*IBUS_MHz);
            break;
        case 0x7:  /*  PLL_CFG = 0b0111  */
            printf("Core Freq(4.5x)=%dMHz & ",(int)(4.5*(float)IBUS_MHz));
            printf("VCO Freq(2x)=%dMHz\n", 9*IBUS_MHz);
            break;
        case 0xb:  /*  PLL_CFG = 0b1011  */
            printf("Core Freq(5x)=%dMHz & ", 5*IBUS_MHz);
            printf("VCO Freq(2x)=%dMHz\n", 10*IBUS_MHz);
            break;
        case 0x9:  /*  PLL_CFG = 0b1001  */
            printf("Core Freq(5.5x)=%dMHz & ",(int)(5.5*(float)IBUS_MHz));
            printf("VCO Freq(2x)=%dMHz\n", 11*IBUS_MHz);
            break;
        case 0xd:  /*  PLL_CFG = 0b1101  */
            printf("Core Freq(6x)=%dMHz & ", 6*IBUS_MHz);
            printf("VCO Freq(2x)=%dMHz\n", 12*IBUS_MHz);
            break;
        case 0x3:  /*  PLL_CFG = 0b0011  */
            printf("PLL in bypass!\n");
            break;
        case 0xf:  /*  PLL_CFG = 0b0011  */
            printf("CLOCK OFF! How can this be???\n");
            break;
        default:
            printf("ERROR - INVALID PLL_CFG!");
    }   /* End of HID1 switch  */
return;
}
```

Conclusions:

Students should be able to note that:

- A 64-bit rotate instruction would be very useful but has to be synthesized in the 32-bit PowerPC architecture.
- The PowerPC Embedded Application Binary Interface (EABI) specifies how arguments are passed to an assembly language routine and values are returned.
- The syntax for assembly language programs varies widely among compiler/assembler vendors.
- The only way to pass data between the integer registers (GPRs) and the floating point registers (FPRs) on PowerPC is by writing and reading to memory.
- With wise use of the register set, the memory access to pass information from the GPRs to the FPRs is the only memory access the assembly language routine needs - thus improving performance.

Troubleshooting:

If the student is not able to:

- Get started. Suggest that the student code the conversion in a c program until they have proven their algorithm. If they are still having difficulty, the disassembly of the c program could provide insight.
- Get the desired returned values from function calls. This is a good opportunity to use breakpoints and examine the registers to determine how the expected value is being returned.

Experiment

10

Dhrystone Benchmarking

Problem Statement:

- In this experiment the student will adapt the popular Dhrystone benchmark to execute on the Excimer board.

Objectives:

Upon completion of this laboratory experience, students will be able to:

- verify a popular industry metric of processor performance in embedded applications, Dhrystone Version 2.1 Vax MIPs, for the PowerPC 603e microprocessor on the Excimer board.
- compare processor performance, as measured by Dhrystone, to published values for other processors.
- compare code generation and instruction scheduling, and resulting performance, for several competing compilers on the Dhrystone benchmark.
- substitute more highly optimized routines for the built-in or library functions provided by compiler vendors to improve performance.
- utilize the dtime function of Experiment 4 to measure elapsed time for a benchmark's execution.

Background Information:

Compararative performance of computers is a popular topic for computer scientists, computer architects, and computer salesmen. Many performance measurements, or benchmarks, have been used over

the last several decades to compare various aspects of computer performance. Some benchmarks involve running real applications, e.g. compiling the compiler or calculating a spreadsheet, which are heavily dependent on the resources of a particular operating system. Others are small synthetic benchmarks designed to be representative of the workload of a class of larger applications but which do no meaningful work and are easier to run across various operating systems and architectures.

The Dhrystone benchmark is a synthetic benchmark developed by Reinhold P. Weicker of Siemens-Nixdorf in the early eighties. It was first published in "Communications of the ACM" vol. 27., no. 10 (Oct. 1984), pp. 1013 - 1030. It is easily ported to many different operating environments and results for many computers are widely published. For embedded processors, where operating system and system resources may be limited, it has been the most often quoted performance measure. It is popular because it provides one number – Vax MIPS – that can be compared quickly with other computers. (Vax MIPS are calculated by dividing the number of times that the Dhrystone benchmark completes in a second by the number of Dhrystones per second performed by the now-ancient Vax 11/780 from Digital Equipment Corporation.) On the other hand, it is widely disparaged because it is so small that it fits entirely within the first level cache of most modern microprocessors and compiler vendors soon made a game out of optimizing it to get ever-higher Vax MIPs numbers.

Motorola publishes Dhrystone 2.1 Vax MIPs numbers for the PowerPC 603e processor on the Excimer board because the number is often requested. After the first loop through the benchmark, it resides entirely in the L1 cache of any PowerPC microprocessor. At that point the performance varies linearly with frequency and the results reflect the efficiency of the micro-architecture and the effectiveness of the compiler in generating code to capitalize on it. Motorola's published numbers are 1.41 Vax MIPs per Mhz. For an Excimer board running at 133Mhz (to keep it comfortably cool in a still air environment), that equates to 188 Vax MIPs.

The Dhrystone benchmark (and numerous others) is available in it's official source code via anonymous ftp to 'ftp.nosc.mil' in directory 'pub/aburto'. The IP address is: 128.49.192.51. Instructions for exe-

cuting the benchmark and "rules" for execution are available there as well. Comparative results for many computers are available from the same site or from various news groups including 'comp.benchmarks'.

Procedure:

1. Download the Dhrystone Version 2.1 benchmark from the ftp site. Read the associated instructions for compilation and execution. You will find that the benchmark calls the C library functions `strcpy` and `strcmp` inside the measurement loop and `printf` and `scanf` outside the measurement loop. You will also find that the benchmark calls a timer function dtime() that returns a count of seconds as a double floating point number. You will need to assemble and link the assembly language code from Experiment 4 which reads the PowerPC time base facility and converts the 64 bit integer value to time in seconds based on Excimer's 66Mhz bus speed.

Reminder: Function calls such as `printf` will have to be equated to `dink_printf` routine to print results to the terminal. Dhrystone also queries the user via the scanf function for the number of times to run the benchmark. A scanf function using DINK's `getchar` and `writechar` will have to be written and substituted or the number of times through the benchmark hard coded. If hard-coding the number of runs, be certain to use a variable instead of a constant, as a constant would change the benchmark inside the measurement loop. Motorola makes no changes to the benchmark which would unfairly change the result when compared to other results.

2. Compile the Dhrystone source code files, link in the dtime() function, and execute the benchmark on the Excimer board. Compare your results to Motorola's published numbers.

Hint: Maximum performance will be obtained only when running entirely out of cache. If the SRAM access LED on Excimer is not out during execution then the program is not running entirely from the internal cache. DINK's regmod command may be needed to enable the instruction and data cache (regmod HID0 to new value of 8000c000). A bug in early versions of DINK may also require modifying the data memory mapping unit (DMMU) to make the data accesses cacheable (regmod rbat1l to new value of 00000012).

3. Examine the disassembled code. The strcmp library function offers one opportunity for performance enhancement. Many C libraries compare strings one byte at a time. Motorola provides a library of highly optimized functions including strcmp on their website at

http://www.mot.com/PowerPC/teksupport. The assembly language for strcmp from this library is shown below. Notice that when possible this function compares strings four bytes – a word – at a time, thus reducing memory (or in this case, cache) accesses by 75%. Assemble and link this strcmp function in place of the stdlib function. Did performance improve?

```
#------------------------------------------------------------------
# Copyright, Motorola, Inc.  All Rights Reserved.  This
# software contains proprietary and confidential information of
# Motorola, Inc.  Use, disclosure or reproduction is prohibited
# without the prior express written consent of Motorola, Inc.
#------------------------------------------------------------------

#------------------------------------------------------------------
# int strcmp(const unsigned char* source1,
#            const unsigned char* source2);
# Returns:
#  value < 0  if source1 < source2
#  value = 0  if source1 = source2
#  value > 0  if source1 > source2
#------------------------------------------------------------------

       .set   _eq,2
       .set   _cr0,0
       .set   _cr1,1

#aix#  .toc
#aix#T..strcmp:
#aix#  .tc    ..strcmp[ tc], strcmp[ ds]
#aix#  .align 2
#aix#  .globl strcmp[ ds]
#aix#  .csect strcmp[ ds]
#aix#  .long  .strcmp[ pr],TOC[ tc0],0
#aix#  .globl .strcmp[ pr]
#aix#  .csect .strcmp[ pr]
#aix#.strcmp:

       .sect  .text
       .align 2
       .extern strcmp
strcmp:

#nt#   .reldata
#nt#   .globl strcmp
#nt#strcmp:
#nt#   .long  ..strcmp,.toc
#nt#   .text
#nt#   .globl ..strcmp
#nt#..strcmp:

       # r0 = temporary
       # r3 = source1 pointer, result, mask for first words
       # r4 = source2 pointer
       # r5 = 0x80808080
       # r6 = 0x01010101
       # r7 = source2 word
       # r8 = source1 word
       # r9 = temporary
       # r10 = source1 pointer
       # r11 = temporary
       # r12 = index
```

```
        # See if the two pointers are both word aligned.
        xor     r0,r3,r4
        rlwinm. r0,r0,0,30,31
        addis   r6,r0,0x0101
        mr      r10,r3
        bne     Byte_By_Byte

        # Generate an initial index so the word containing the first byte
        # will be loaded.  Compute a mask to set all bits in the bytes
        # prior to the first in the words that are loaded.
        rlwinm  r11,r3,3,27,28
        li      r3,-1
        rlwinm  r12,r10,0,30,31
        subfic  r11,r11,32
        neg     r12,r12
        slw     r3,r3,r11

        # Complete the setup for the word aligned loop.
        ori     r6,r6,0x0101
        lwzx    r8,r12,r10
#le#    lwbrx   r8,r12,r10
        or      r8,r8,r3        # Mask off unused bytes.
        slwi    r5,r6,7
        subfc   r0,r6,r8
        andc    r9,r5,r8
        lwzx    r7,r12,r4
#le#    lwbrx   r7,r12,r4
        and.    r11,r0,r9
        addi    r4,r4,-4
        addi    r12,r12,4
        or      r7,r7,r3        # Mask off unused bytes.
        bne     Source1_Has_Null

Word_Loop:
        subfc.  r3,r8,r7
        bne     Words_Differ
        lwzx    r8,r12,r10
#le#    lwbrx   r8,r12,r10
        subfc   r0,r6,r8
        andc    r9,r5,r8
        and.    r11,r0,r9
        addi    r12,r12,4
        lwzx    r7,r12,r4
#le#    lwbrx   r7,r12,r4
        beq     Word_Loop

Source1_Has_Null:
        # We terminated the loop because r8 has a null byte.
        # Shift both words right so the null byte is the LSB.
        # Can't do this with cntlzw because of a borrow if the byte
        # preceeding the null has the value one.
        rlwinm. r10,r8,0,0,7
        li      r9,24
        beq     shift
        rlwinm. r10,r8,0,8,15
        li      r9,16
        beq     shift
        rlwinm. r10,r8,0,16,23
        li      r9,8
        beq     shift
        li      r9,0
shift:
        srw     r7,r7,r9
```

```
        srw     r8,r8,r9
        subfc   r3,r7,r8
        blr

Words_Differ:
        # We terminated the loop because the words differ but
        # r8 does not have a null byte.  Return 1 or -1 based
        # on the unsigned comparison.
        subfe r3,r3,r3
        nand r3,r3,r3
        ori r3,r3,1
        blr

Byte_By_Byte:
        # Do strcmp a byte at a time.
        lbz     r9,0(r3)
        lbz     r0,0(r4)
        subfc.  r3,r0,r9
        bnelr
Byte_Loop:
        cmpi    _cr1,0,r9,0
        beq     _cr1,Null_Byte
        lbzu    r9,1(r10)
        lbzu    r0,1(r4)
        subfc.  r3,r0,r9
        beq     Byte_Loop
        blr
Null_Byte:
        mr      r3,r0
        blr
```

References:

[1] "Communications of the ACM" vol. 27., no. 10 (Oct. 1984), pp. 1013 - 1030.

Suggested Code:

```
/*  File - "dry1.h"
 *  Defines the functions defined in support.c and used by dhry21a.c
 */


#define  printf my_printf
#define fprintf my_fprintf
#define fopen my_fopen
#define fclose my_fclose
#define exit my_exit
extern int my_printf(const char *, ...);
extern int my_fprintf(const char *, ...);
extern FILE * my_fopen();
extern int my_fclose();
extern void my_exit();
```

```
/* file "support.c"
 * This file provides substitutes for some of the library function calls
 * used in Dhrystone which DINK doesn't support.
*/
/* Set the number of dhrystone loops here.  */
#define NUMBER_OF_RUNS 10000000

#include <stdarg.h>

/* These are the magic addresses for the DINK functions.  */
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6638;

/* A version of malloc that will only supply up to 2048 bytes total.  */
char *malloc(unsigned int size)
{
  static char buffer[2048];
  static char *next = buffer;
  char *p = next;
  next += ((size + 7) & ~7);
  if (next >= buffer + sizeof(buffer))
    /* Terminate by executing a zero.  */
    asm(".long 0");
  return p;
}

/* Scanf is used only to read the number of times through the loop.  */
/*ARGSUSED*/
void scanf(char *fmt, int *v)
{
  *v = NUMBER_OF_RUNS;
}


/* This only will handle the printf calls in dhrystone.  The DINK printf
   doesn't work for floating point, so convert the value here to integer.  */
int my_printf(const char *fmt, ...)
{
  int a1, a2, a3, sign;
  char *neg_zero_fmt;
  double round, fraction, val;
  va_list ap;
  va_start (ap, fmt);
  if (strcmp(fmt, "%7.1lf \n") == 0) {
```

```
    fmt = "%6d.%1d \n";
    neg_zero_fmt = "    -%1d.%1d \n";
    round = 0.05;
    fraction = 10.0;
    goto fake_float;
  } else if (strcmp(fmt, "%10.1lf \n") == 0) {
    fmt = "%9d.%1d \n";
    neg_zero_fmt = "       -%1d.%1d \n";
    round = 0.05;
    fraction = 10.0;
    goto fake_float;
  } else if (strcmp(fmt, "VAX MIPS rating = %10.3lf \n") == 0) {
    fmt = "VAX MIPS rating = %9d.%03d \n";
    neg_zero_fmt = "VAX MIPS rating =        -%1d.%03d \n";
    round = 0.0005;
    fraction = 1000.0;
  fake_float:
    val = va_arg(ap, double);
    if (val < 0) {
      sign = -1;
      val = -val;
    } else {
      sign = 1;
    }
    /* Round the value.  */
    val += round;
    a1 = val;
    a2 = val * fraction - a1 * fraction;
    if (a1 == 0 && sign == -1)
      fmt = neg_zero_fmt;
    a1 *= sign;
  } else {
    a1 = va_arg(ap, int);
    a2 = va_arg(ap, int);
    a3 = va_arg(ap, int);
  }
  dink_printf(fmt, a1, a2, a3);
  va_end (ap);
}

/* Dummy out the calls to exit, fopen, fprintf, and fclose.  */
void my_exit() {}
int my_fopen() {  return 1; }
```

```
int my_fprintf() {}
int my_fclose() {}
```

Conclusions:

Students should be able to note that:

- "There are lies, damn lies, and benchmarks, in that order".
- The Dhrystone benchmark is small enough to understand, see opportunities for optimization, and port easily to various computer environments.
- The Dhrystone benchmark is string intensive and the resulting performance metric may be meaningless in applications involving other workloads, e.g. extensive mathematical calculations or bit manipulation.
- Not all compilers are created equal. The sample compilers shipped in the Excimer kit may generate vastly different code, instruction scheduling, and results on this benchmark. However, the biggest impact probably comes from the Motorola hand-coded strcmp function. Like most vendors, Motorola strives to provide the best benchmark results possible for marketing reasons..

Troubleshooting:

If the student is not able to:

- Get a time function. The suggested code for Experiment 4 provides a `double dtime(double TICS_PER_SECOND)` function which can be easily modified to provide timing information for this benchmark.
- Link with the DINK supplied `printf` or other functions. Check the addresses for the respective functions in DINK using the symtab command.
- Get the results to print from the dhry21a.c program. dink_printf will not accept floating point formats. The results will have to be typecast as unsigned long or int to print. The loss of accuracy is insignificant.

Experiment

11

Running the Linpack Benchmark (Debugging Stage)

Problem Statement:

- In this experiment the student will adapt the popular Linpack benchmark to execute on the Excimer board. (Contributed by Walter Guiot and Luis Narváez).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- verify a popular industry metric of processor performance in embedded applications, Linpack, for the PowerPC 603e microprocessor on the Excimer board.
- compare processor performance, as measured by Linpack, to published values for other processors.
- compare code generation and instruction scheduling, and resulting performance, for several competing compilers on the Linpack benchmark.
- compare processor performance, as measured by Linpack, with performance given by the manufacturer.

Background Information:

LINPACK is a collection of subroutines used to benchmark the performance of computers in the analysis and solving of linear equations and linear least-squares problems. LINPACK solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. The LINPACK routines are constructed such that locality of reference is maximized.

A C language version can be obtained: http://www.netlib.org/benchmark/ .

References:

[1] David A. Patterson, John L. Hennessy, Conputer Organization & Design The Hardware / Software Interface Morgan Kaufmann Publishers, Inc San Francisco, California 1994.

[2] http://www.netlib.org/linpack/

[3] The Linpack Benchmark: http://www.netlib.org/benchmark/top500/reports/report93/section2_16_2.html

Procedure:

1. Download the linpack benchmark from ftp sites such as http://www.netlib.org/linpack or ftp://ftp.nosc.mil/pub/aburto.
2. Read the included documentation regarding compaling instruction.
3. Modify the source code to make it compatible with Dink instructions such as dink_printf. Remember that dink_printf does not support floating point, a function will have to be developed to handle floating point, refer to experiment #5 for printf function for the Dhrystone benchmark.
4. Compile the source code and link it with the dtime() function develop in experiment #?. The file dtime.s developed in the experiment may be used.
5. Get the PowerPC 603e performance information from the manufacturer and compare it with the results obtained with linpack.

Suggested Code:

This experiment is still in the ebugging phase as it is necessary to send floating point numbers to th screen.

```
!file "dtime.s"  (For Metware High C/C++ Compiler/Assembler)
!  Assembly language routine to convert 64-bit PowerPC TB facility to
!  Double-precision, floating-point number.  (Plus additional routines for
!  testing.)   CJC 981216 (Contributed by Chuck Corley, Motorola)
!   Register usage:
!          r3 = FPU       (upper 32 bits of floating point value)
!          r4 = FPL       (lower 32 bits of floating point value)
!          r5 = TBU(time base upper - read from spr or loaded for test)
!          r6 = TBL(time base lower - read from spr or loaded for test)
!          r7 = leading zeroes in a register or shift count of +/-(zeroes - 11)
!          r8 = accumulator for final EXPonent value of DPFP number
!          r9 = shift count of 32 - n where n = +/-(zeroes -11)
!          r10 = constant register of 11
!          r11 = link register storage

#define TBU 269;          !Special purpose register numbers for TB
#define TBL 268;
    .data
Local_storage:
    .double 0
    .text
    .global dtime
    .global get_HID1
    .global conversion_test
!For CodeWarrior:
!asm double conversion(double TICS)
conversion:
    cntlzw  r7,r5          !Find leading zeroes in TBU. Preserve in r7.
    addi    r9,r0,32          !Will need a 32 in several places. Create one in r9.
    addi    r10,r0,11      !Create a constant in r10 = 11.
    subf.   r8,r7,r9          !r8 will hold EXP.  Currently (32 - leading zeroes)
    beq+    tbu_is_zero    !TBU never got incremented? (Zeroes=32?) (Most likely)

    subf.   r7,r10,r7      !No. Is TB more than 2^^52?  (Zeroes<11?) r7 = (Z-11)
    add     r8,r8,r9          !Final exponent will be (64 - 1 - leading zeroes).
    bge+    tbu_lt_8yrs    !If TB>2^^52, shift TBU bits right. (Not likely)

tbu_gt_8yrs:               !for Z<11: fpu = tbu>>n=(11-Z);
                                    !fpl = tbu<<n=(32-(11- Z))|tbl>>n=(11-Z);
    neg     r7,r7             !rlwnm shift count of (11-Z) = -(Z-11) = n = r7.
    subf    r9,r7,r9          !rlwnm shift count of 32-n = 32 - (11-Z) = r9.
    rlwnm   r3,r5,r9,12,31 !Shift TBU right n = (11 - Z). Mask off [ 0:11] .
    rlwnm   r4,r5,r9,0,10  !Shift remaining TBU bits left n = 32-(11-Z)
    rlwnm   r6,r6,r9,0,31  !Shift TBL right n = (11 - Z)
    or             r4,r6,r4          !Or rest of TBU shifted left with TBL shifted right.
    b       form_exponent  !Go bias the exponent and or into FPU.

tbu_lt_8yrs:               !for Z>=11: fpu=tbu<<n=(Z-11)|tbl>>n=(32-(Z-11));
                                    !fpl=tbl<<n=(Z- 11);
    subf    r9,r7,r9       !Form a shift count of 32 - (11-Z) = r9.
    rlwnm   r3,r5,r7,12,31 !Shift TBU left n = (Z-11). Mask off [ 0:11] .
    srw     r5,r6,r9       !Shift TBL bits right n = 32-(Z-11).
    or             r3,r3,r5   !Or TBU shifted left with TBL shifted right.
    rlwnm   r6,r6,r7,0,31  !Shift remainder of TBL left n = (Z-11).
    xor     r4,r6,r5       !XORing with the same value shifted right is like ANDing
```

```
    b      form_exponent  !fpl with a mask of all zeroes in bits [ 32-(Z-11):31] .

tbu_is_zero:              !Z= 32
    cntlzw r7,r6          !Find leading zeroes in TBL.
    subf.  r8,r7,r9           !EXP = (32 - leading zeroes).
    beq-   tbl_is_zero    !Entire TBL count exactly zero?  (Not likely)
    subf.  r7,r10,r7      !No. Is TB less than 2^^20?  (zeroes < 11?)
    bge-   tbl_lt_63ms    !If not, will have to shift bits right. (Most likely)

tbl_gt_63ms:              !for z<11: fpu = tbl>>n=(11-z); fpl = tbl<<n=(32-(11-z));
    neg            r7,r7      !rlwnm shift count of (11-Z) = -(Z-11) = n = r7.
    subf   r9,r7,r9           !rlwnm shift count of 32-n = 32 - (11-Z) = r9.
    rlwnm  r3,r6,r9,12,31 !Shift TBL right n = (11 - z). Mask off [ 0:11] .
    rlwnm  r4,r6,r9,0,10  !Shift remaining TBL bits left n = 32 - (11 - Z).
    b      form_exponent

tbl_lt_63ms:              !for z>=11: fpu = tbl<<(z-11); fpl = 0;
    rlwnm  r3,r6,r7,12,31 !Shift TBL left n = (Z-11). Mask off bits 0-11.
    xor    r4,r4,r4           !fpl = 0.
    b      form_exponent

tbl_is_zero:              !for Z=32 && z=32: fpu = fpl = 0;
    xor    r3,r3,r3           !Unlikely result that TB was zero. Prepare to
    xor    r4,r4,r4           !return all zeroes for the floating point value.
    b compute_seconds

form_exponent:
    addi   r8,r8,1022     !Add DP bias (1023) -1 to the exponent
    rlwinm r8,r8,20,1,12  !Biased DP EXP will be (63-(leading zeroes in TB)+1023).
    or     r3,r3,r8

compute_seconds:
    lis    r5, Local_storage@h
    ori    r5, r5, Local_storage@l
    stw    r3, 0(r5)
    stw    r4, 4(r5)
    lfd    f2, 0(r5)      !Load back in as 64bit float
    fdiv   f1,f2,f1           !Divide by bus clock ticks per second
    blr                              !Return time in seconds as double in fp1

!  Routine passed sample values of TBU and TBL.  Returns FPU and FPL as
!  unsigned long.
!For CodeWarrior:
!asm struct TB_View * conversion_test(unsigned long Upper,
!                                    unsigned long Lower, double TICS)
conversion_test:
    or             r5,r3,r3           !Use test values of TBU and TBL passed in r3 and r4
    or             r6,r4,r4           !as substitutes for values read from TB.
    mflr   r11                !Save the return address.
    bl             conversion !Convert TBU and TBL into FPU and FPL
    mtlr   r11                !Return in r3 and r4
!For CodeWarrior:
!   la             r3,Local_pointer(SP)!Return a pointer to the FPU storage location.
    blr

!  Routine passed sample values of TBU and TBL.  Returns seconds as double.
!For CodeWarrior:
!asm double float_test(unsigned long Upper, unsigned long Lower, double TICS)
float_test:

    or             r5,r3,r3           !Use test values of TBU and TBL passed in r3 and r4
    or             r6,r4,r4           !as substitutes for values read from TB.
    mflr   r11                !Save the return address.
    bl             conversion !Convert TBU and TBL into FPU and FPL
```

```
    mtlr    r11                !Return as double in fpr1
    blr

! Routine reads the TBU and TBL. Returns seconds as double.
!For CodeWarrior:
!asm double dtime(double TICS)
dtime:
read_TB:
    mfspr   r5,TBU          !Get TBU.
    mfspr   r6,TBL          !Get TBL.
    mfspr   r7,TBU          !Get TBU again.
    subf.   r7,r5,r7            !Did it increment between reading TBU and TBL?
    bgt-    read_TB         !If so, read them again.  (Not likely)
    mflr    r11                 !Save the return address.
    bl      conversion      !Convert TBU and TBL into FPU and FPL
    mtlr    r11
    blr

! Routine reads the HID1 (PLL_CFG) register. Returns in r3.
get_HID1:
    mfspr r3,1009           !Get HID1 register.
    blr
```

Conclusions:

Troubleshooting:

1. Make sure you are using the correct address for the dink_printf function.
2. Make necessary changes to handle floating point.
3. Use the timer funciton develop in experiment #?, do not use any of the ones provided with the benchmark.

Experiment

12

Cache Impact on Benchmark Metrics (Debugging Stage)

Problem Statement:

- In this experiment the student will compare the Linpack benchmark results with cache memory disabled to the results with cache memory enabled. (Contributed by Walter Guiot and Luis Narváez).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- compare processor performance, as measured by Linpack, with and without cache enabled.
- understand the advantages of cache memory in a computer system.

Background Information:

Cache memory is a special type of random access memory (RAM) that stores the most recently used instructions and/or data from a larger main memory system. Cache memory can be accessed faster than regular RAM.

Cache memory is categorized in levels. Level I (L1) cache memory is on the same chip as the microprocessor. Level II (L2) and later levels are usually separate memory chips. The microprocessor first looks for the instructions in L1 cache, if it is not there (a miss) it goes to the next level, it continues looking from level to level until reaching main memory or in the worst case a mass storage device, such as disk drives or hard drives. These memory chips are typically static RAM (SRAM) modules that do

not need to be electromagnetically refreshed as DRAM does. These characteristics make cache memory faster and more expensive than regular RAM.

There is a slight catch with cache memory, if there is a cache miss, then it takes around more clocks cycle to access data from DRAM, or ROM. For this reason a L2 cache that is too small could theoretically decrease performance.

The 603e provides independent 16-Kbyte, four-way set-associative instruction and data caches. The cache line is 32 bytes in length. The caches use a least recently used (LRU) replacement policy.

The caches provide a 64-bit interface to the instruction fetch unit and load/store unit. The surrounding logic selects, organizes, and forwards the requested information to the requesting unit. Write operations to the cache can be performed on a byte basis, and a complete read-modify-write operation to the cache can occur in each cycle. The load/store and instruction fetch units provide the caches with the address of the data or instruction to be fetched.

Procedure:

1. Develop an assembly program to enable and disable cache memory. Refer to the PowerPC 603e manual for the registers involve in enabling and disabling the cache.
2. Follow the procedure of experiment #? “Running the Linpack Benchmark”.
3. Link the assembly code developed here to the code of experiment #?.
4. Run the benchmark with cache enable and note the results.
5. Disable cache and run the benchmark again, compare both results and state your conclusions.

Questions:

References:

[1] The L2 Company: What is Cache? http://www.mindspring.com/~l2co/WhatIsCa.html

[2] What is…cache memory? http://www.whatis.com/cachemem.htm

[3] MPC603e & EC603e RISC Microprocessors User's Manual

Suggested Code:

Troubleshooting:

1. Make sure you are using the correct registers for enabling and dissabling cache.
2. Refer to the troubleshooting to experiment #? "Running the Linpack benchmark".

Experiment

13

Flash ROM (Debugging Stage)

Problem Statement:

- This experiment requires the development of an assembly language program starting on Programmer Space RAM location $70000 that will copy a program which resides on RAM location $71000 to a free space Flash ROM location. Program copied into Flash ROM will auto-execute from its present location. (Contributed by José I. Quiñones and Eisen Montalvo-Ruiz).

Objectives:

Upon completion of this laboratory experience, students will be able to:

- Write and assemble an assembly language subroutine.
- Execute a piece of code that will in turn copy another piece of code to Flash ROM and execute it.
- Write assembly code directly into Flash ROM space by means of assembly code.

Background Information:

Any microprocessor-based system needs memory devices to hold data and program instructions. Memories can be classified as volatile and non-volatile.

The basic difference between both realms of memories is that volatile memories looses its data contents when power is removed from the semiconductor chip while non-volatile memories holds its data contents even when power is removed. This has some very interesting implications, which must remain clear to microprocessor based system designers since both types of memory have their advantages and disadvantages as well as a typical use.

Non volatile memory is used when a system is to execute a dedicated application. Take for example your computer. When you turn it on, it executes a self-initialization procedure we call “booting”. How does the CPU know what to do? The BIOS (Binary Input Output System) is the dedicated application for initialization and is stored on a non-volatile type of memory. If this program is by any means erased, the computer will just never be able to restart!

Once the computer starts, the application to be executed can be anything we decide. It would be quite expensive, and space prohibiting, to have all the applications we would like to have on a computer stored on non-volatile silicon memory chips. Instead we have found quite useful to store our applications on magnetic or other type of media and then write them to a bank of volatile type of silicon memory, which is fast and can cope with the microprocessor need for instructions to execute.

This volatile memory can be written over and over repeatedly. And when power is no longer applied to the memory array, all information is forever lost. Typical use of volatile memory, (such as RAM) is to load Operative Systems (OS) and any application you may want to execute on the computer. Only the necessary instructions reside on RAM. The OS is responsible of loading the RAM with the necessary instructions as they are needed.

A disadvantage of non-volatile memory cells is that they have a short life and usually can not be rewritten more than a specified number of times. They are also significantly slower than volatile memories.

We have come to accept that there is a need and a use for both types of memory. This is why our Excimer board is equipped with 1 MB of RAM, which is our volatile type of memory and 4MB of Flash ROM, which is our non-volatile memory.

At this time you must be familiar with the fact that the Excimer board has an “operative system” you can use, called the DINK32. This Monitor program is stored on the Flash ROM and is responsible of

initializing all peripheral activity within the Power PC 603e evaluation board. When you press reset, this dedicated application executes and it takes over the board. This monitor also enables the user to see and use all registers and memory space with commands such as Memory Display (MD), Register Display (RD) and so on. The Monitor even has an online assembler and disassembler that enables the user to see code and to enter code manually.

The Excimer also needs RAM to operate. That is, any variable and/or data must be written to RAM since this value may change continuously. So the 1 MB of RAM provided is needed by the Excimer to operate. Fortunately for developers, this RAM can also be loaded with applications trying to exploit the power embedded on the 603e CPU. For testing and evaluation purposes, the RAM will hold instructions (which must be downloaded periodically with the help of a PC) that can be traced and watched using the DINK32 tools.

It is the ultimate goal of any Engineer using an evaluation board such as the Excimer, to create a free running application capable of self-supporting itself. In other words that an specified application such a control or embedded system may run without the need of a PC computer. This implies that the Engineer code will always be present on the Excimer Board but it was only downloaded once.

You are currently downloading the code every time you need to execute it. Eventually you will reach a time when your code will be totally debugged. It would be really appropriate to make the Evaluation Board a stand-alone unit with your code as the dedicated application.

It is the goal of this laboratory experiment to show how to write the Flash ROM area so that a dedicated application may be coded in this non-volatile memory. The architecture of the suggested procedure is to write a simple program that writes another code into the Flash ROM. As a more advanced option, the “writer program” could set the RESET vector to the address where it will begin writing the “written code”.

As a safety feature, so that DINK32 is still available after the flash ROM is updated, the "written code" is to ask the user if the application to be executed is DINK32 or itself. The true DINK32 pointer can be saved from the previous vector table. The program will now have the ability to either jump to the short code (which can be the LED blinker code) or to the DINK32.

How to program the Flash ROM:

The Excimer board has 4Mbytes of Flash ROM where the DINK32 monitor resides. Nevertheless, user defined applications can be recorded on this memory space as long as some precautions are taken. Do recall that if the monitor is no longer working, successive Flash recording might not be as easy (or possible with existing hardware).

The Flash ROM chips being interfaced by the Power PC 603E on our Excimer board are AMD's AM29LV800B 8 Mbits memory modules. Detailed information on how to erase and program the Flash cells can be found in AM29LV800B Data Sheet. Some introductory information follows, but it is advised to students to read the proposed Application Note.

The AMD FLASH ROM chip already contains a control unit inside of each FLASH chip. All that is needed to erase, read or program a byte, sector or the entire chip is a set of commands which will put the chip into a predefined state. Once the state is defined and the corresponding commands sent to the chip via the Data Bus, the internal control logic will do the rest. States which can be entered are: Sector Protect, Sector Unprotect, Autoselect, Erase Sector, Erase Chip, Erase Suspend, Erase Resume, Program, Reset and Unlock Bypass

Device programming occurs by executing the Program Command sequence. This initiates the **Embedded Program** algorithm—an internal algorithm that automatically times the program pulse widths and verifies proper cell margin. The **Unlock Bypass** mode facilitates faster programming times by requiring only two write cycles to program data instead of four. Instead of using the common Program Command

(6 step command) you can now use the Unlock Bypass Command Sequence (refer to Table 5 of the AM29LV800B Data Sheet).

Device erasure occurs by executing the Erase Command sequence. This initiates the **Embedded Erase** algorithm—an internal algorithm that automatically preprograms the array (if it is not already programmed) before executing the erase operation. During erase, the device automatically times the erase pulse widths and verifies proper cell margin.

The host system can detect whether a program or erase operation is complete by observing the RY/BY# pin, or by reading the DQ7 (Data# Polling) and DQ6 (toggle) **status bits**. After a program or erase cycle has been completed, the device is ready to read array data or accept another command. The **sector erase architecture** allows memory sectors to be erased and reprogrammed without affecting the data contents of other sectors. The device is fully erased when shipped from the factory.

The AM29LV800B Data Sheet explains all other mentioned states. Also how to enter and exit the mentioned modes of operation can be clearly seen on Tables 4 and 5 of said document. But be careful. although AMD is very specific in telling all addresses where specific data has to be written, the implementation of the Excimer board did changed these parameters.

The way the four Flash ROM chips were assembled on the Excimer Board (configured as a two memory bank of double 16 bit words) redefines all addresses and expected data to and from the memory chips. First of all is the definition of the Power PC 603E Microprocessor data bus as big-endian while the Flash ROM data bus is a little-endian device. This means that what is the MSB to the Microprocessor is actually the LSB to the Flash ROM! Data has to be bit reversed (and that is as simple as mirroring the 16 bit words) so that the Flash ROM understands. In other words whatever data the AM29LV800B Data Sheet tells you to send to the Flash, has to be bit reversed before it is actually sent.

NOTE: This is actually the programmer's job. Programmer could develop a subroutine to mirror the word or else he/she could do it by hand before actually writing the assembly or C code.

The other very important fact to have in mind is that the memory mapping suggests a 3 bit left shifting. That is (if observing the Excimer implementation schematic) the 3 less significant bits are not used to select memory space inside the memory device itself (recall that PowerPC address bus is big-endian while AMD flash devices are little-endian. That is why address lines A31, A30 and A29 are the ones not connected to the memory device. They are actually the less significant address lines to the Power PC 603e). These three address lines are in fact used by the Excimer memory control FPGA to select one of the four memory chips. That is why on the Excimer implementation Schematic there is a different line for each one of the WE* (write enables) control signals while CS* (Chip Select) and OE* (Output Enable) are shared.

NOTE: Since the three less significant address lines are used only for chip selection while programming, every address present on the AM29LV800B Data Sheet (including command sequence as well as sector segregation) has to be shifted left by 3.

The following are the required rules for the Excimer V1 and V2 boards since the address line is shifted by three and the data lines are bit reversed.

- Rule for converting from expected address (found on AM29LV800B Data Sheet) to shifted address (Excimer Memory Mapped) left shift address by 3, bit reverse data example:

  address 0x555 < 3 = 0x2aa8 (0b0101 0101 0101 < 3 = 0b0010 1010 1010 1000)

- Rule for converting from bit not reversed (little Endian to Little Endian) to bit reversed (Little Endian to Big Endian). Bit reverse the data line example:

  data 0xaa = 0x55 (0b1010 1010 = (bit reversed) 0b0101 0101)

On the following sequences, address and data has already been shifted as well as bit reversed. You should not have trouble if using these examples. Do recall that you will need to make reference to the AM29LV800B Data Sheet when trying to search for sectors and specific addresses.

Entering Autoselect mode

Write address: 0x2aa8 with data: 0x55555555
Write address: 0x1550 with data: 0xaaaaaaaa
Write address: 0x2aa8 with data: 0x09090909

Get Manufacturer ID:

Read address: 0x0000 get data: 0x80008000

Get Device ID:

Read address: 0x0008 get data: 0x5B445B44

Get Sector Protect status for each sector:

Read address: 0x[SA] get data: 0x00000000 for non protected
Read address: 0x[SA] get data: 0x80008000 for protected

Reset sequence exit autoselect mode:

Write address: 0x0000 with data: 0x0f0f0f0f

Erasing a flash sector sequence:

Write address: 0x2AA8 with data: 0x55555555
Write address: 0x1550 with data: 0xAAAAAAAA
Write address: 0x2AA8 with data: 0x01010101
Write address: 0x2AA8 with data: 0x55555555
Write address: 0x1550 with data: 0xAAAAAAAA
Write address: sector address with data: 0x0C0C0C0C

Programming flash memory:

Write address: 0x2AA8 with data: 0x55555555
Write address: 0x1550 with data: 0xAAAAAAAA
Write address: 0x2AA8 with data: 0x05050505
Write address: Word address with data to be programmed

Now some notes: The Flash ROM device can change a one to a zero in any cell bit, but not visa versa. Thus, it is a good practice to erase the memory first before writing to it. Otherwise after writing, the memory may be corrupted, since a zero can't be changed back to a one. In some cases you can play with the byte to be recorded. If for example a cell has the byte 0x55 programmed on it and you want to

write 0x11, it can be done without erasing the cell. This is a good idea when there are only a few bytes to be programmed, but it would not be wise when programming large amounts of data.

Erasing can only be done one sector at a time or the entire chip, so one can not erase only a portion of a sector. The smallest amount of flash to erase is one complete sector. (Refer to Table 3 on the AM29LV800B Data Sheet for sector addresses and remember to shift left by 3 any address). Be careful when erasing a sector. There might be important code (as OS code) on the sector.

References:

- AM29LV800B Data Sheet (www.amd.com)
- FL.C Source Code

Suggested Code:

Two sets of code are available on this section. The first thing any student should try is to properly send commands to the Flash ROM chips. Since writing or erasing may be hazardous to the Excimer health, it is recommended that a few experiments are performed before any erasing or programming is attempted. This will allow the students to practice Assembly Language Programming to interface the ships not by using the data supplied by the AM29LV800B Data Sheet but with the already shifted addresses and inverted data.

The first code snippet shows how to enter Autoselect Mode. Students will be able to check memory spaces for the specified data. If any method to memory display byte information shows the specified data (note inversion will be noted) the command sequence has been successful. Otherwise, one or more steps might not be correct (Check troubleshooting section for more information on possible errors).

```
        .text
        .global readflash
readflash:
        xor r12, r12, r12           !Clearing Registers 12 and 13
        xor r13, r13, r13
```

```
        lis r12, 65472              !R12 contains $FFC00000
        addi r12, r12, 8                    !Register12 contains address $FFC00008
                                    !This address is used to request the Autoselect mode data as
                                    !specified by the Flash ROM data sheet + a shift by 3.

        xor r14, r14, r14           !Clearing Register 14
        lis r14, 65472              !Register 14 contains $FFC00000
        addi r14, r14, 10920        !Register 14 contains $FFC02AA8
                                    !This address is used to send the first step into the
                                    !Autoselect sequence as specified by the Flash ROM data
                                    !sheet + a shift by 3.

        xor r15, r15, r15           !Clearing Register 15
        lis r15, 65472              !Register 15 contains $FFC00000
        addi r15, r15, 5456         !Register 15 contains address $FFC01550
                                    !This address is used to send the second step into the
                                    !Autoselect sequence as specified by the Flash ROM data !sheet
                                    + a shift by 3.

        xor r16, r16, r16           !Clearing Register 16
        addi r16, r16, 170          !Through this steps, the data $AAAAAAAA which will be
        slwi r16, r16, 8            !sent to the Flash on the first step of the Autoselect
        addi r16, r16, 170          !sequence, is assembled. Note that bit reversal has been
        slwi r16, r16, 8            !accounted for.
        addi r16, r16, 170
        slwi r16, r16, 8
        addi r16, r16, 170          !Register 16 contains data $AAAAAAAA

        xor r17, r17, r17           !Clearing Register 17
        lis r17, 21845              !Data $55555555 is assembled through this steps. This is
        addi r17, r17, 21845        !the data that will be sent on the second step of the
                                    !Autoselect sequence. Bit reversal has been taken care of.

        xor r18, r18, r18           !Clearing Register 18
        addi r18, r18, 9            !Data $09090909 is assembled through this steps. This
        slwi r18, r18, 8            !word is sent to the Flash to differentiate the command
        addi r18, r18, 9            !sequence from all others. This data is specific to the
        slwi r18, r18, 8            !Autoselect mode.
        addi r18, r18, 9
        slwi r18, r18, 8
        addi r18, r18, 9            !Register 18 contains data $09090909

        stwx r17, r14, r13          !Store $55555555 data in $FFC02AA8
        addi r14, r14, 4            !First sequence step taken care of.
        stwx r17, r14, r13          !It has to be written to both memory banks.

        stwx r16, r15, r13          !Store $AAAAAAAA data in $FFC01550
        addi r15, r15, 4            !Second sequence step taken care of.
        stwx r16, r15, r13          !It has to be written to both memory banks.

        subi r14, r14, 4            !Subtract 4 from R14 so that the address FFC02AA8 is
                                    !once again available.
        stwx r18, r14, r13          !Store $09090909 data in $FFC02AA8
        addi r14, r14, 4            !Third step into the Autoselect sequence taken care of.
        stwx r18, r14, r13          !It has to be writen to both memory banks.

    ! At this moment, the Flash ROM is on the Autoselect Mode. All memory reads will return
!Autoselect Mode Information such as Manufacturer ID, Device ID and Sector Protect
!Verification status. To exit Autoselect Mode, the Autoselect Reset sequence or a Hardware
!reset must be performed.

        lwzx  r19, r12, r13         !Loading into Register 19 Manufacturer ID for the first
                                    !memory bank.
        addi r12, r12, 4
```

```
        lwzx  r20, r12, r13          !Loading into Register 20 Manufacturier ID for the second
                                     !Memory Bank
```

The second snippet of code is actually the subroutine needed to write a byte. Extreme care must be taken when writing data to the Flash as Excimer can easily become corrupt and inoperating. Students are encouraged to practice on a free sector (preferably sector 16), which will certainly be erased, or else can be erased without fear of damaging the OS. Check the AM29LV800B Data Sheet for further Excimer sector division information. Refer to the troubleshooting section for problems regarding difficulty to write data to the Flash.

```
        .text
        .global writeflash
writeflash:

        xor r13, r13, r13            !Clearing Register 13

        xor r12, r12, r12            !Clearing Register 12
        lis r12, 65472               !Register 12 contains Address $FFC00000
        addi r12, r12, 10920         !Register 12 contains Address $FFC02AA8

        !xor r14, r14, r14           !Clearing Register 14
        !lis r14, 65472                     !Register 14 contains Address $FFC00000
        !addi r14, r14, 10924        !Register 14 contains Address $FFC02AAC

        xor r15, r15, r15            !Clearing Register 15
        lis r15, 65472               !Register 15 contains Address $FFC00000
        addi r15, r15, 5456          !Register 15 contains Address $FFC01550

        xor r16, r16, r16            !Clearing Register 16
        addi r16, r16, 170           !Through this steps, the data $AAAAAAAA which will be
        slwi r16, r16, 8             !sent to the Flash on the first step of the Autoselect
        addi r16, r16, 170           !sequence, is assembled. Note that bit reversal has been
        slwi r16, r16, 8             !accounted for.
        addi r16, r16, 170
        slwi r16, r16, 8
        addi r16, r16, 170           !Register 16 contains data $AAAAAAAA

        xor r17, r17, r17            !Clearing Register 17
        lis r17, 21845               !Data $55555555 is assembled through this steps. This is
        addi r17, r17, 21845         !the data that will be sent on the second step of the
                                     !Autoselect sequence. Bit reversal has been taken care of.

        xor r18, r18, r18            !Clearing Register 18
        addi r18, r18, 5             !Data $05050505 is assembled through this steps. This
        slwi r18, r18, 8             !word is sent to the Flash to differentiate the command
        addi r18, r18, 5             !sequence from all others. This data is specific to the
        slwi r18, r18, 8             !Autoselect mode.
        addi r18, r18, 5
        slwi r18, r18, 8
        addi r18, r18, 5             !Register 18 contains data $05050505

        stwx r17, r12, r13           !Store $55555555 data in Address $FFC02AA8
        addi r12, r12, 4             !First sequence step taken care of.
        stwx r17, r12, r13           !It has to be written to both memory banks.
```

```
        stwx r16, r15, r13          !Store $AAAAAAAA data in $FFC01550
        addi r15, r15, 4            !Second sequence step taken care of.
        stwx r16, r15, r13          !It has to be written to both memory banks.

        subi r12, r12, 4            !Subtract 4 from R12 so that the address FFC02AA8 is
                                    !once again available.
        stwx r18, r12, r13          !Store $05050505 data in Address $FFC02AA8
        addi r12, r12, 4            !Third sequence step taken care of.
        stwx r18, r12, r13          ! It has to be written to both memory banks.

        xor r19, r19, r19           !Clearing Register 19
        lis r19, 65484              !Register 19 contains Address $FFCC0000
        xor r20,r20,r20             !Clearing Register 20
        lis r20, 85                 !Register 20 contains data $00000055

        stwx r20, r19, r13          !Program $00000055 data in FLASH Address $FFCC0000

!At this moment, data will have been programmed into $FFCC0000

/*-------------------------------------------------------------------------------------*

main.c

EMR 29/4/99
This code writes a program to FlashROM and sets the booting vector to the program written.
There's some problem with the FlashROM. Right now the program is stuck with trying to write cor-
rectly to the FlashROM. I have tried the same program with two different Excimer Boards and the
results are different. Although the second time it write some information correctly. I think  the
FlashROM is damaged.

-------------------------------------------------------------------------------------*/
#include <stdio.h>

#define  getchar dink_get_char
#define  putchar dink_write_char
#define  printf dink_printf

void blink_leds(int addr, int i);
unsigned long (*dink_get_char)() = (unsigned long (*)()) 0x1e4c4;
unsigned long (*dink_write_char)(char) = (unsigned long (*)(char)) 0x5eb4;
unsigned long (*dink_printf)() = (unsigned long (*)()) 0x6270;

#define flash      0xffc00000

#define addr1      0x2aa8
#define addr2      0x1550

#define zerosones  0x01010101
#define allas      0xaaaaaaaa
#define allfives   0x55555555
#define zeroscs    0x0c0c0c0c
#define zerosfives 0x05050505
#define zerosfours 0x04040404
#define zerosnines 0x09090909
#define allzeros   0x00000000

void erase_sector(int);
void program(unsigned int *, unsigned int);
void unlock_bypass();
void write_word(unsigned int *, unsigned int);
void unlock_bypass_reset();
void leds_main();
void blink_leds(int addr, int i);
void blank();
```

```
void main()
{
	unsigned int *addr = (unsigned int *)0xffd00000;
	unsigned int *prog = (unsigned int *)0x0;
	unsigned int size  = 0;

	//Borrar sector donde estara el codigo
	erase_sector(7);

	//Escribir nuestro codigo al FlashROM
	/*unlock_bypass();
	size = (unsigned int)blank - (unsigned int)leds_main;
	for(unsigned int i=0; i<size/4; i++)
	{
		prog = (unsigned int *)((unsigned int)leds_main + i*4);
		printf("%x ", (unsigned int)prog);
		//program(addr+i, *prog);
		write_word(addr+i, *prog);
	}
	unlock_bypass_reset();*/

	prog = (unsigned int *)((unsigned int)leds_main);
	write_word(addr, *prog);

	//Traer 1er sector de MDink al RAM

	//Cambiar Boot Pointer al vector table

	//Borrar 1er sector del MDink

	//Reescribir el sector al FlashROM

	//Rebootear
}

void erase_sector(int sector)
{
	unsigned int *sect, *command;

	sect=0;
	if(sector<4)
	{
		//Sector 3 or less
		switch(sector)
		{
			case 0:
				sect = (unsigned int *)(0x0 + flash);
				break;
			case 1:
				sect = (unsigned int *)((0x2000<<3) + flash);
				break;
			case 2:
				sect = (unsigned int *)((0x3000<<3) + flash);
				break;
			case 3:
				sect = (unsigned int *)((0x4000<<3) + flash);
		}
	}
	else if(sector>18)
	{
		return;
	}
	else
```

```
        {
                sect = (unsigned int *)((0x8000<<3)*(sector-3)+flash);
        }

        command  = (unsigned int *)(flash + addr1);
        *command = allfives;
        command +=1;
        *command = allfives;

        command  = (unsigned int *)(flash + addr2);
        *command = allas;
        command +=1;
        *command = allas;

        command  = (unsigned int *)(flash + addr1);
        *command = zerosones;
        command +=1;
        *command = zerosones;

        command  = (unsigned int *)(flash + addr1);
        *command = allfives;
        command +=1;
        *command = allfives;

        command  = (unsigned int *)(flash + addr2);
        *command = allas;
        command +=1;
        *command = allas;

        printf("%x ", (unsigned int)sect);

        *sect    = zeroscs;
}

void program(unsigned int *addr, unsigned int word)
{
        unsigned int *command;

        command  = (unsigned int *)(flash + addr1);
        *command = allfives;
        command +=1;
        *command = allfives;

        command  = (unsigned int *)(flash + addr2);
        *command = allas;
        command +=1;
        *command = allas;

        command  = (unsigned int *)(flash + addr1);
        *command = zerosfives;
        command +=1;
        *command = zerosfives;

        printf("%x ", (unsigned int)addr);
        printf("%x\n", word);
        *addr = word;
}

void unlock_bypass()
{
        unsigned int *command;

        command  = (unsigned int *)(flash + addr1);
        *command = allfives;
```

```
        command +=1;
        *command = allfives;

        command  = (unsigned int *)(flash + addr2);
        *command = allas;
        command +=1;
        *command = allas;

        command  = (unsigned int *)(flash + addr1);
        *command = zerosfours;
        command +=1;
        *command = zerosfours;
}

void write_word(unsigned int *addr, unsigned int word)
{
        unsigned int *command;

        command  = (unsigned int *)(flash);
        *command = zerosfives;
        command +=1;
        *command = zerosfives;


        printf("%x ", (unsigned int)addr);
        printf("%x\n", word);
        *addr = word;
}

void unlock_bypass_reset()
{
        unsigned int *command;

        command  = (unsigned int *)(flash);
        *command = zerosnines;
        command +=1;
        *command = zerosnines;

        command  = (unsigned int *)(flash);
        *command = allzeros;
        command +=1;
        *command = allzeros;
}

void leds_main()
{
        int decimal_no;
        char LED;
        char number;
        do
        {
                printf ("\nSelect the LED you want to blink:\n");
                printf ("\tS - Press S for the Status LED\n");
                printf ("\tE - Press E for the Error LED\n");
                printf ("\tQ - Press Q to Quit\n");
                LED = getchar();
                if (LED == 'E' || LED == 'e')
                {
                        printf ("\nEnter the number of times (1-9) to blink the Error LED: ");
                        do{
                                number = getchar();
                        }while ( !((number >= '0') && (number <= '9')) );
                        putchar(number);
                        decimal_no = number - 48;
```

```
                        blink_leds(0x40600000, decimal_no);

                }
                else if (LED == 'S' || LED == 's')
                {
                        printf ("\nEnter the number of times (1-9) to blink the Status LED: ");
                        do{
                                number = getchar();
                        }while ( !((number >= '0') && (number <= '9')) );
                        putchar(number);
                        decimal_no = number -48;
                        blink_leds(0x40200000, decimal_no);
                }
        } while ( LED != 'Q' && LED != 'q' );       /*  X or x */
    return;
}


void blink_leds(int addr, int i)
{
        unsigned long count;
        int loop;
        for (loop = 0 ; loop < i; loop++)
        {
                *(char *) (addr) = 0x00;     //turn on error
                for(count = 0; count <= 0xfff00; count ++);
                *(char *) (addr) = 0x08;     //turn off error
                for(count = 0; count <= 0xfff00; count ++);
        }
        *(char *) (0x40600000) = 0x08;
}

void blank(){}
```

Troubleshooting:

Trouble to enter a mode using the specified word sequence tends to occur either because data was not properly reversed or because address was not properly shifted. Recall that the memory mapping of the Excimer does not have to be that of the AMD memory devices as specified on the AM29LV800B Data Sheet. In fact no memory device has to actually be memory mapped as specified by a datasheet. Those addresses provided by the manufacturer are offsets and depend greatly on were they were placed.

On the Excimer Board, Flash ROM was placed on address $FFC00000. Every offset has to be added to that base address. But this is not all. Since Power PC data bus is 64 bits wide and chips are just 16, they must be cascaded to supply the need. If you are using the AM29LV800B Data Sheet as reference, take in mind that every address presented has to be shifted by three. If you are using the notes presented on this lab section, the addresses are already shifted.

As explained earlier, Power PC 603e is a big-endian device while AMD Flash chips are little endian. Address bus was already fixed so that little address bit were reversed by hardware. Data bus does not has to be hardware reversed since it is irrelevant if you programmed a cell backwards. When it is read, it will get backwards again and thus rectified. This leaves programmers with the problem that whatever the Flash chip is expecting as a command will have to be reversed by hand. Bit reversing is sometimes confused as taking a "1" and changing it to a "0" and viceversa. This will actually not work when writing commands to the Flash device.

What we mean by bit reversing is actually a mirror of the word. That what was the MSB now becomes the LSB and viceversa. If you are having trouble entering into a specific mode check that you have done this right. Another common mistake is to reverse the nibbles in the bytes or the bytes on the word. You actually have to reverse the 16 bit word. An "0A" reversed is not "A0" but "50". (00001010 mirrored is not 10100000 but it is 01010000).